Efficient Agreement Protocols for Asynchronous Distributed Systems

Izabela Moise

N^o d'ordre: 4481 **ANNÉE 2011**

THÈSE / UNIVERSITÉ DE RENNES 1
sous le sceau de l'Université Européenne de Bretagne

pour le grade de

DOCTEUR DE L'UNIVERSITÉ DE RENNES 1

Mention: INFORMATIQUE
Ecole doctorale MATISSE

présentée par

Izabela Moise

préparée à l'unité de recherche n^o 6074 - IRISA
Institut de Recherche en Informatique et Systèmes Aléatoires
IFSIC

Efficient Agreement Protocols in Asynchronous Distributed Systems

Thèse soutenue à Rennes
le 12 decembre 2011

devant le jury composé de:

Michel HURFIN / directeur de thèse
Chargé de recherche, INRIA Rennes, France

Jean-Pierre LE NARZUL / encadrant
Maitre de conférence, Télécom Bretagne Rennes, France

Roberto BALDONI / rapporteur
Professeur, Sapienza Università Di Roma, Rome, Italie

Sébastien TIXEUIL / rapporteur
Professeur, LIP6, Paris, France

César VIHO / président
Professeur, Université de Rennes 1, Rennes, France

Achour MOSTÉFAOUI / examinateur
Professeur, Université de Nantes, Nantes, France

"First you guess. Don't laugh, this is the most important step. Then you compute the consequences. Compare the consequences to experience. If it disagrees with experience, the guess is wrong. In that simple statement is the key to science. It doesn't matter how beautiful your guess is or how smart you are or what your name is. If it disagrees with experience, it's wrong. That's all there is to it."
– Richard Feynman

"If we knew what it was we were doing,
it would not be called research, would it?"
– Albert Einstein

Acknowledgments

This manuscript would not have seen the light of day without my advisor, Michel Hurfin. I am most grateful for his support, patience and guidance throughout my PhD work. I am very fortunate to have had the opportunity to work under his supervision and my gratitude goes beyond professional aspects. A great amount of thanks are for Jean-Pierre Le Narzul, my second advisor. I was lucky enough to work with people from which I could learn a lot and in a very pleasant manner, due to their great sense of humor. Thank you both for everything.

Kind regards go as well to the members of the ADEPT team who made the working environment a very pleasant one. Many thanks to my "second" team also, the members of KerData team.

I am very grateful to Prof. César Viho for presiding over my thesis defense and also to the members of the jury, Prof. Roberto Baldoni, Prof. Sébastien Tixeuil, Prof. Achour Mostéfaoui for evaluating my work and providing useful feedback.

Many thanks to all my friends here in Rennes. Their names are too many to be mentioned here. A simple "thank you" does not seem enough to express my gratitude for the help of my girls, my sister Diana and Alexa. Thank you, les filles.

Abstract

The Consensus problem is recognized as a central paradigm of fault-tolerant distributed computing. In a purely asynchronous system, Consensus is impossible to solve in a deterministic manner. However, by enriching the system with some synchrony assumptions, several solutions have been proposed in order to circumvent the impossibility result, among which the Paxos approach introduced by Lamport.

This work represents a contribution to the construction of efficient consensus protocols in asynchronous distributed systems. The algorithmic contribution of this thesis consists of an efficient framework, called the Paxos-MIC protocol, that follows the Paxos approach and integrates two existing optimizations. Paxos-MIC generates a sequence of consensus instances and guarantees the persistence of all decision values. The main feature of the protocol is its adaptability. As one of the optimizations may be counterproductive, Paxos-MIC incorporates a triggering mechanism that dynamically enables the optimization. This mechanism relies on several triggering criteria with the main purpose of predicting if the activation of the optimization will be beneficial or not. Extensive experimentations on the Grid'5000 testbed were carried out with the purpose of evaluating the protocol and the efficiency of the triggering criteria.

A second part of this work focuses on the use of consensus as a building-block for designing higher-level applications. We consider the particular context of transactional mobile agents and we propose a solution to support the execution of transactions in an ad-hoc network. This solution relies on an everlasting sequence of decision values, generated by repeatedly invoking a consensus building-block. The consensus service is provided by the Paxos-MIC framework.

Contents

Chapter 1

Introduction

Contents

DISTRIBUTED systems are everywhere. They emerged for two main reasons: inherent distribution (applications that require sharing of resources or dissemination of information among geographically distant entities are "natural" distributed systems) and also as a solution to satisfy requirements such as fault-tolerance. Indeed, a general approach for supporting fault-tolerance consists in decentralizing and replicating components, functions and data. Variations at the level of synchrony properties, inter-process communication, failure model of the system and many other factors, have led to a prolific literature revolving around distributed systems.

The construction of distributed systems rises many challenges, among which the main challenge remains: how do we coordinate the components of a distributed system? Replication requires a protocol that enables replicas to agree on values and actions. Many relevant practical problems, such as atomic broadcast for state-machine replication or decentralized atomic commit for database transactions, can be reduced to the problem of reaching *Consensus*.

Consensus encapsulates the inherent problems of building fault-tolerant distributed systems. Consensus represents the greatest common denominator of the so-called class of *Agreement problems* such as data consistency, group membership, consistent global states, distributed consensus, atomic broadcast and many others.

1.1 Context

In the general context of system models, we address the particular settings of asynchronous distributed systems prone to crash failures and message omissions. In this context, providing efficient solutions to agreement problems is a key issue when designing fault-tolerant applications. The state machine approach [55] illustrates this concern. In this particular example, replicas of a critical server need to agree on a sequence of incoming requests. Such a sequence is usually constructed by repeatedly calling a Consensus service. This importance explains why Consensus has attracted so much interest. The classical specification of the Consensus problem [42] requires that each participant proposes an initial value and, despite failures, all the correct processes decide on a single value selected out of these proposals. Solving consensus goes back three decades ago, with a fundamental result in distributed computing: the *FLP impossibility result* stating that Consensus is unsolvable by a deterministic algorithm in an asynchronous system even if only a single process may crash [42]. The intuition behind this result is that without any synchrony assumptions, there is no reliable way of detecting failures, in other words, we do not know if a process is really crashed or just slow. Nevertheless, under some well-identified additional synchrony properties that can be indirectly exploited by a failure detector or a leader-election service, deterministic consensus protocols have been proposed.

In the context of algorithms, we refer to two major contributions: the $\Diamond S$ based protocols, in particular the *CT* protocol [16], and the *Paxos* [37, 38] protocol. Both approaches solve consensus in an asynchronous system extended with an *oracle* that provides useful information for the process that invokes it. $\Diamond S$ based protocols rely on an iterative control structure: processes proceed in a sequence of asynchronous rounds, with the main purpose of allowing a convergence towards a same value and eventually decide on it.

The classical rigid interaction scheme requires that each participant to the consensus, provides an initial value and then waits for the decision value. Many solutions have been proposed to bypass this rigid scheme. One of the most famous is the *Paxos* protocol [37, 38]. Indeed, in Paxos-like protocols, a participant to a consensus instance is neither required to propose an initial value nor to wait for a returned decision value. This is achieved by identifying roles that participants may fulfill. A *proposer* is a process able to supply an initial value during a consensus instance. A *learner* is a process interested in acquiring a decision value. In an open system, their number is unbounded and may vary in time. However, it is assumed that at least one non-crashed proposer supplies an input during each consensus instance. In Paxos-like protocols, proposers and learners are external entities and they are not involved in the consensus mechanism, which is driven only by the interaction between *acceptors* and *coordinators*.

We aim at understanding the underlying principles of both approaches and also the similarities and the differences between them. Although they rely on different types of oracles, namely failure detectors and leader-election services, the two approaches have strong similarities: the agreement property of consensus is never violated regardless of asynchrony and messages losses. In addition, both are based on the same "most recent voting scheme".

Once solving consensus became understood, a hot topic quickly emerged: how can the performance of a consensus algorithm be improved? This interest was mainly motivated by the use of consensus as a building block for higher-level applications. Indeed, many techniques used in distributed systems often rely upon a sequence of decision values, usually

generated by repeatedly invoking a consensus service. The performance of the application is directly impacted by the efficiency of the underlying consensus protocol. A traditional cost criterion for consensus algorithms is defined by the number of communication steps required to reach a decision in a well-behaved run of the algorithm. This is called the *latency*. In particular, a prolific research trend has explored techniques for improving the performance of Paxos-like protocols.

In this general context, the objectives of our research were twofold. Our core goal was to design and evaluate a protocol able to efficiently build an everlasting sequence of decisions. Second, we wanted to outline the interest of using such a building-block in applications based on transactional mobile agents.

1.2 Contributions

The main contributions of this thesis can be summarized as follows:

Formal definition of the Multiple-Integrated Consensus problem. This work focuses in a first phase, on both the construction and the availability of an everlasting sequence of decision values, during a long-lasting computation. We denote the problem of generating such a sequence by *the Multiple-Integrated Consensus problem*. We assume that this sequence is created, step by step, by a rather stable subset of dedicated processes called *the core*. This subset is in charge of generating the sequence of decisions and also it can provide all the decision values already computed (or the most recent ones) to any interested process. Members of the core interact with external entities that act as proposers or as learners and also they cooperate among themselves to establish the sequence of decisions. We extend the classical formal definition of the Consensus problem in order to specify properties when coping with a sequence of consensus instances.

Paxos-MIC: an adaptive fast Paxos for making quick everlasting decisions. The algorithmic contribution of this work consists in designing an efficient framework that is adaptive, reaches fast decisions and ensures the persistence of all decision values. This protocol, called *Paxos-MIC*, satisfies the specification of the Multiple-Integrated Consensus problem. The Paxos-MIC protocol allows to solve a sequence of consensus instances in an unreliable, asynchronous system. As suggested by its name, the protocol follows the Paxos approach (the Paxos protocol and several of its variants [37, 38]). Our contribution proceeds in several phases. We begin by revisiting the interaction scheme between proposers, learners, coordinators and acceptors. We concretize this interaction in an architecture describing the communication pattern between all entities involved. Further, we provide an algorithmic description of the Paxos-MIC protocol, sustained by a detailed description of the underlying principles of the protocol. Paxos-MIC integrates two optimizations: a safe optimization (denoted $\mathbf{S_O}$) that is always activated and a risky optimization (denoted $\mathbf{R_O}$) that is activated at runtime, only in favorable circumstances. The main feature of the framework is its *adaptability*: for each consensus instance, the leader checks at runtime if the context seems suitable for activating optimization $\mathbf{R_O}$.

We try to have a presentation of the Paxos-MIC protocol that remains as close as possible to the terminology and the basic principles used in Paxos [38] and Fast Paxos [39]. Never-

theless, some choices that have conducted the design of our protocol, may help the reader to have a slightly different look at all the protocols belonging to the Paxos Family. Thus, this manuscript is also a didactic contribution, as it may help to get a better understanding of different contributions, by merging them within a single simple framework.

Interest of activating the risky optimization. Part of our work was dedicated to investigating in which conditions the risky optimization would lead to a performance gain. To meet this goal, we analyzed the performance of the protocol and observed its behavior in several specific scenarios. During the evaluation phase of the Paxos-MIC protocol, we proceed first with identifying a series of synthetic benchmarks. We consider the behavior of the protocol when either only $\mathbf{S_O}$ or both $\mathbf{S_O}$ and $\mathbf{R_O}$ are used. Our aim is to determine the impact of some contextual factors (size of the core, geographical position of the actors) on the time required to reach a decision and also to obtain an assessment of the performance degradation of optimization $\mathbf{R_O}$ when used in less favorable circumstances.

Prediction of collisions. One of the main benefits of Paxos-MIC consists in the dynamic activation of optimization $\mathbf{R_O}$ at runtime. This optimization is risky and may lead to significant degradation of the protocol when proposers provide different values for the same consensus instance. In this case, we say that a *collision* occurs. The question that arises is the following: is it possible to predict a collision? In our approach, the leader is in charge of deciding the activation of optimization $\mathbf{R_O}$, at runtime, between two consecutive consensus instances. Its decision is based upon a so-called *triggering criterion*. The specification of this criterion may allow us to answer the aforementioned question. Based on the performance analysis of Paxos-MIC, we define four reference contexts that are relevant for optimization $\mathbf{R_O}$. We propose several triggering criteria. Their definition relies on different knowledge, such as an analysis of the recent past, the study of the current context and also a possible prediction of the future. In order to obtain a comparison and an assessment of their efficiency, we consider a particular application (a secure Web server) and a real trace that records the activation dates of the successive consensus instances. Through an analysis of the trace, we evaluate the expected gain in each of the four contexts and we observe the accuracy of the triggering criteria in predicting future collisions.

Consensus as a building block. Consensus is intensively used as a building block for constructing higher-level applications. Generating an everlasting sequence of decisions is at the heart of the state machine approach. Usually this approach is used to maintain consistency between replicas. We consider a particular application where the sequence of decisions aims at implementing a reliable server that supervises the behavior of transactional mobile agents. We propose a cloud-based solution to support the execution of transactions in an ad-hoc network. Mobile agents migrate among the places of an ad-hoc network with the goal of building an itinerary that is able to commit a given transaction. We identify two services, *Availability of the Source* and *Atomic Commit*, that provide support for the agent execution. These services ensure reliability and atomic validation of the transaction and can be supplied by entities located in a cloud. These higher-level services are constructed by relying on an everlasting sequence of decisions, generated by repeatedly invoking a consensus service. We propose a solution where these two services are provided in a reliable and ho-

mogeneous way. To guarantee reliability, the proposed solution relies on a single agreement protocol that orders continuously all the new actions whatever the related transaction and service. The agreement service is provided by the Paxos-MIC protocol. The two identified services require reaching agreement on three types of decisions: the source of the itinerary, the itinerary itself and the outcome of this itinerary. We show how such an architecture can be constructed by relying on Paxos-MIC as a building-block.

Implementation and evaluation. An important part of this work has been dedicated to providing an efficient practical implementation of the Paxos-MIC protocol, based on the algorithmic description. A great amount of work has also been invested in the evaluation of the protocol, through a series of synthetic benchmarks. All experiments involved in the aforementioned contributions were carried out on the Grid'5000/ALLADIN experimental testbed federating 10 different sites in France. It is an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS, RENATER and other contributing partners. We are particularly grateful for the excellent support that was provided by the Grid'5000 team during the time in which the work presented in this thesis was carried out.

1.3 RoadMap

This manuscript is organized in five main parts.

The first part: Context - Consensus in asynchronous distributed systems. This part of the manuscript provides a general overview of the context of our work. This part contains two main chapters that present the state of the art of distributed asynchronous consensus protocols. Chapter 2 presents the consensus problem in asynchronous distributed systems and an important result obtained regarding this problem. Further, this chapter focuses on failure detectors and in particular, explores deterministic consensus protocols that are based on $\Diamond S$ failure detectors.

In a second step, Chapter 3 introduces the Paxos approach, as a famous and efficient solution to the consensus problem. After describing the underlying principles of the protocol, we focus on variants of Paxos, proposed with the purpose of optimizing the protocol's latency. The motivation of these protocols comes from the repeated use of consensus protocols as building-blocks for higher-level application, among which the state machine approach represents a powerful example.

The second part: Contribution - Paxos-MIC, an adaptive fast Paxos. The second part of the manuscript presents the algorithmic contribution of this thesis: the development of *Paxos-MIC* - an efficient framework for making quick everlasting decisions. Chapter 4 commences with the formal specification of the *Multiple Integrated Consensus* problem that allows us to identify the context of our approach and the motivation for proposing our protocol. We briefly discuss the system model and also the architecture of our protocol from the point of view of the entities involved and the communication scheme. The presentation of the Paxos-MIC protocol is done in two steps. We describe a first version of the protocol that only inte-

grates optimization $\mathbf{S_O}$ (used in FastPaxos [8]). Then, we present the modifications required to use (depending on the context) both optimizations. This two-step description allows to clearly identify the parts of the protocol that are impacted by optimization $\mathbf{R_O}$ (introduced by Lamport in Fast Paxos [39]).

The third part: Implementation and evaluation. In the third part, we discuss the implementation details of the algorithm introduced in Chapter 4. Chapter 5 discusses the implementation of the protocol in two steps: firstly, it provides an overview of the event-based architecture, EVA and later, in a second step, we show how abstractions defined in EVA are used as building blocks for the Paxos-MIC protocol. This chapter also insists on software engineering aspects and other practical issues and technical details that we encountered.

In a second phase, this part evaluates the implementation described in Chapter 5. Chapter 6 is structured in two main parts: first, the Paxos-MIC protocol is evaluated in this through a series of synthetic benchmarks. These benchmarks consist of specific scenarios that facilitate the study of the protocol's performance and the analysis of its behavior. In a second step, Chapter 6 focuses on the risky optimization $\mathbf{R_O}$. As this optimization is unsafe and can lead to additional cost when used in bad circumstances, its activation is carefully done at runtime by relying on the so-called *triggering criterion*. We propose several triggering criteria and we evaluate them in the context of a particular application (a secure Web server).

The Paxos-MIC protocol combines two optimizations dynamically. It allows to solve several consensus instances and to guarantee the persistence of all decision values. The protocol is adaptive as it tries to obtain the best performance gain depending on the current context. Between two consecutive consensus instances, the Leader determines if the optimization $\mathbf{R_O}$ has to be triggered or not. We analyzed its behavior when both optimizations (or just the safe one $\mathbf{S_O}$) are used. We studied favorable and unfavorable scenarios where $\mathbf{R_O}$ may lead to an additional cost. We considered an application for securing a Web server and showed that simple triggering criteria do not allow to predict accurately collisions but they are precise enough to adapt dynamically the behavior of the protocol to the current context. All our results demonstrate that there is a real interest in using $\mathbf{R_O}$.

The fourth part: Sequence of decisions as a support for transactional mobile agents. of the manuscript focuses on using the Paxos-MIC protocol as a building-block for higher-level applications. Chapter 7 addresses the context of transactional mobile agents and proposes a cloud-based solution to support the execution of transactions in ad-hoc networks. First, this solution identifies two important services that provide support for the agent execution. Further, these higher-level services are constructed by relying on a sequence of decision values generated by the repeated invocation of a consensus service. This service is provided by the Paxos-MIC protocol, executed by processes located in a cloud. We define the interface that manages the interactions between the Paxos-MIC building-block and the higher-level services that invoke it.

The fifth part: Achievements and perspectives is represented by Chapter 8 and summarizes the aforementioned contributions and presents a series of future perspectives that are interesting to explore.

Part I

Context: Consensus in Asynchronous Distributed Systems

Chapter 2

Consensus in Asynchronous Distributed Systems

Contents

« *A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.* »

Leslie Lamport

This informal way of defining a distributed system captures one of the most important features of a distributed system: despite its distributed nature, the system behaves as a single coherent entity, from the point of view of the users. The definition of Leslie Lamport also insists on the fact that a distributed system is prone to failures. A distributed system is defined as a collection of independent computing entities, located on different computers. These entities (represented by software and hardware components) are sometimes also called *processes*.

2.1 System model

In a distributed system, the entities are connected through a network and can be viewed as nodes in a connected graph where the edges are the communication links. Processes perform local computation and coordinate their actions by exchanging relevant information. The system model is defined by a set of assumptions characterizing the properties of the system. The communication mechanism used by the entities is part of the specification of the system model. In a shared memory model, processes communicate by accessing shared objects located in a common memory. In a second category, communication is provided by message passing, i.e., by sending and receiving messages over point-to-point or multicast communication channels.

The system specification also depends on the failure model. By definition, a process that deviates from its execution specification is considered to be *faulty* or *incorrect*. Otherwise, the process is said to be *non-faulty* or *correct*. In the *crash failure model*, a process may fail only by prematurely stopping its execution. A more general failure model is the *Byzantine* failure, defined by an arbitrary behavior of the faulty process: failing to send or receive a message, corrupting local state, or sending an incorrect reply to a request.

In this work, we consider a distributed system prone to crash failures. We define the system as a finite set Π of n processes, with $n > 1$, more precisely $\Pi = \{p_1, p_2, ..., p_n\}$. Different computing units are used to execute the n processes in the system. Among the n processes we assume that at most f processes may fail only by crashing. We also make the assumption that there exists a majority of correct processes, which means that $f < n/2$. As mentioned before, processes communicate by message passing. We assume that each pair of processes is connected by bidirectional, fair lossy channels. The network links are unreliable: messages can be lost and duplicated but not corrupted. However, if a process p_i sends infinitely often a message to a process p_j, then p_j receives infinitely often the message.

In the context of distributed systems, introducing asynchrony usually entails the lack of any global timing assumption. Processes operate at arbitrary speed and there exists no upper bound on message transfer delays. On the contrary, a synchronous model, implies timing bounds on the actions performed by the processes and on message transfer delays.

The asynchronous model is a weak one, but probably the most realistic one. Indeed, in many real systems, there exist arbitrary delays that can be easily modeled by an asynchronous system. In a pure asynchronous system, there is no reliable way of detecting failures. In a synchronous system, a straightforward and reliable solution for failure detection relies on timeout mechanisms. However, in a pure asynchronous model, the question that arises is the following: is a process really crashed or just slow in its computation? Although

the asynchronous model is better suited to represent a real system, a number of important properties have been proven impossible to fulfill, even under very weak assumptions, while they become achievable in the synchronous model.

Failure detection represents an important issue in designing fault-tolerant distributed systems. Failure detection mechanisms usually rely on augmenting a system $\Pi = \{p_1, p_2, ..., p_n\}$ by allowing each process p_i to access a local module, called *a failure detector*. This component outputs the list of processes it currently suspects of being crashed. In other words, it monitors a subset of the processes in the system and updates a list of crashed processes. If a process p_i suspects another process p_j at a given time t, then $p_j \in suspected_i$ at time t, where $suspected_i$ denotes the list provided by the failure detector of process p_i.

However, any failure detector module can be unreliable by providing misleading information. For instance, it can erroneously suspect a process of being crashed, even though the process is behaving correctly, or it can fail to suspect a process that has really crashed. If a failure detector makes mistakes, the list of suspected processes is repeatedly modified and the same process may be added and removed from the list. As a consequence of the unreliability of failure detectors, the lists provided by two different modules located at two different processes may not contain the same entries.

More formally, failure detectors are defined by two types of abstract properties, namely *Completeness* and *Accuracy*. Accuracy specifies which processes are not suspected and when, while completeness specifies which processes are suspected and when. Both properties are needed to avoid a trivial failure detector that will satisfy any accuracy property by never suspecting any process or completeness property by always suspecting all processes. Indeed, a failure detector can make two types of mistakes: it may fail to suspect a faulty process or it may falsely suspect a correct one. Both types of properties aim at limiting the number of mistakes a failure detector can make. More precisely, the completeness requirement limits the number of mistakes a failure detector can make, by not suspecting processes that are really crashed. A failure detector that has the property of completeness eventually suspects every process that actually crashes. Accuracy limits the number of erroneous suspicions that a failure detector can make.

2.2 Consensus

2.2.1 The problem specification

In an asynchronous distributed system prone to crash failures, providing efficient solutions to agreement problems is a key issue when designing fault-tolerant applications. The *State Machine* approach [55] illustrates this concern. In this particular example, replicas of a critical server need to agree on a sequence of incoming requests. The goal is to define a total order on all events that may affect the state of the running replicas. Such a sequence is usually constructed by repeatedly calling a *Consensus* service.

Consensus is recognized as one of the most fundamental problems in distributed computing. The classical specification of the consensus problem [42] requires that each participant to the consensus proposes an initial value (also called, a *proposal* or a *proposed value*) and, despite failures, all the correct processes have to decide on a single value selected out of these proposals. More formally, consensus is defined by the following three properties:

1. **Agreement**: No two correct processes decide on different values.
2. **Termination**: Every correct process eventually decides.
3. **Validity**: Any decided value is a proposed value.

Any algorithm that satisfies these three properties is said to solve the consensus problem. Thus, in more details, consensus is defined by two *safety* properties (*Validity* and *Agreement*) and one *liveness* property (*Termination*). Note that Agreement refers only to correct processes. However, the possibility that a process decides on a different value just before crashing, still exists. In order to prevent this, the stronger notion of *Uniform Consensus* defines the Agreement requirement in the following manner:

1. **Agreement**: No two processes (correct or not) decide on different values.

2.2.2 Agreement problems

Consensus protocols are intensively used as building blocks for implementing higher-level services, of key importance when designing fault-tolerant distributed applications. Many practical problems such as electing a leader, or agreeing on a value of a replicated server, are solved by relying on a consensus algorithm. The consensus problem can be viewed as the "greatest common denominator" for a class of problems known as *Agreement problems*, which includes, among other: Atomic Broadcast, Group Membership and Atomic Commit.

In the following, we provide a general overview of each of these agreement problems.

Atomic Broadcast Reliable Broadcast is the weakest type of fault-tolerant broadcast. It requires that all processes agree on the delivered messages. Informally, if a correct process broadcasts a message then all correct processes eventually receive that message. Reliable Broadcast does not impose any message delivery ordering.

On the contrary, Causal Broadcast guarantees that if the broadcast of a message m_1 happens before (or causally precedes) the broadcast of message m_2, then no correct process delivers m_2 before m_1.

Atomic Broadcast [16] is an agreement problem that entails two aspects: processes agree on the set of messages they deliver and also on the order in which these messages are delivered. Therefore, Atomic Broadcast (sometimes called Ordered Reliable Broadcast) represents a communication paradigm that ensures that all correct processes deliver the same sequence of messages.

More formally, Atomic Broadcast is specified by the *Total Order* requirement: if two correct processes p_i and p_j deliver two messages m_1 and m_2, then p_i delivers m_1 before m_2 if and only if p_j delivers m_1 before m_2.

Group Membership As replication is a key solution for introducing fault-tolerance in distributed applications, group-based computing facilitates the design of such applications. *A group* is defined as a set of processes that cooperate in order to accomplish a common task. If we consider an unbounded set of processes $\Pi = \{p_1, p_2, ..., p_n, ...\}$, any subset forms a group. A composition of a group evolves in a dynamical manner: crash failures of group members

may occur, a process may want to join the group or a member of the group may want to leave it. The current composition of the group defines *the current view* of the group. In group membership, consensus is often used to agree on the sequence of views of the group.

Atomic Commit The *transaction* concept is widely recognized as a powerful model to structure distributed applications. A distributed transaction is structured as a sequence of elementary operations, performed at multiple sites, terminated with a request to commit or abort the transaction. The sites that performed the operations, execute an atomic commit protocol to decide whether the transaction is aborted or committed. The atomic commit protocol implements an *all-or-nothing* approach: either the transaction is committed if and only if each site agrees to commit the specific operation it has executed, or the transaction is aborted on every site.

Executing a distributed task usually comes down to executing a sequence of elementary requests, with the constraint that all of them are viewed as a single operation. Atomic Commit requires that all operations are executed successfully or all of them aborted, if at least one of the operations cannot be completed. Therefore, Atomic Commit can be seen as an agreement problem as it entails agreeing on the outcome of a distributed task: abort or commit the task, as a whole.

2.2.3 Impossibility result

What exactly makes Consensus hard to solve?

When the only failures considered are process crashes, this problem has relatively simple solutions in synchronous distributed systems. However, solving consensus in a purely asynchronous distributed system prone to crash failures is far from being a trivial task. Let us consider a simple scenario described in Figure 2.1: a system consisting of three processes that communicate by message passing. A very simple protocol has the following behavior: each process broadcasts its own value and gathers the messages from the other two processes. Each process then determines the minimum value among the gathered values, and decides on it. Such a protocol, called the *strawman protocol*, would work only if all processes were correct, albeit limited in speed by the slowest process or link.

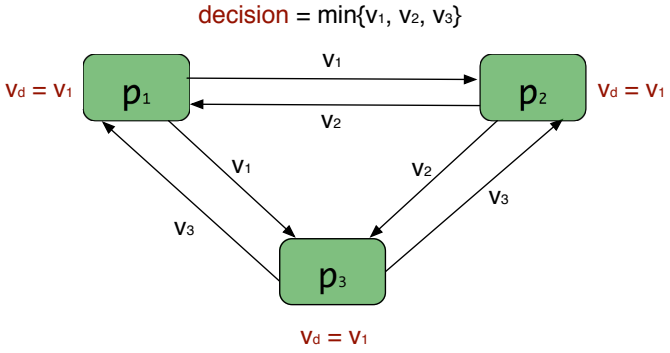


Figure 2.1: Strawman Protocol (best case scenario: all processes are correct).

However, in the presence of a single failure, the processes might gather different sets of values and the minimum among them might not be the same for each process. If processes use timeouts, then each of them might use different timeout on different sets of proposals which would lead to processes deciding on different values, therefore violating the Agreement property. Thus, each process should wait until it has received a value from each other process. But if only one process is faulty and crashes, every other correct process would then wait forever and may never decide on a value (see Figure 2.2).

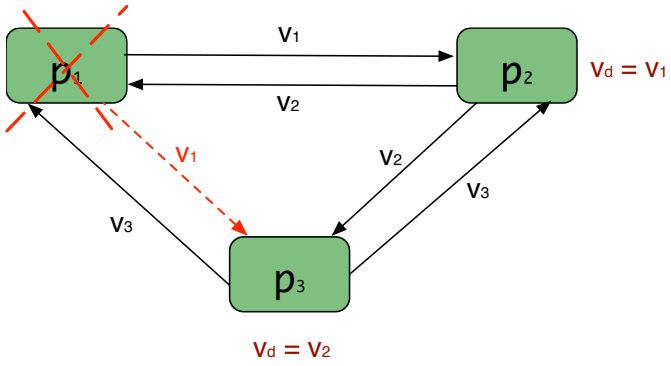


Figure 2.2: Strawman Protocol (when a crash failure occurs).

Solving Consensus in a synchronous system

In a synchronous setting, a straightforward implementation of the strawman protocol relies on a timeout mechanism. In such a context, there exist timing bounds on execution steps and communication latency, a straightforward implementation of the strawman protocol relies on a simple timeout mechanism. It is possible to tune the timeouts to ensure that any suspected process is really crashed. Processes can proceed in simultaneous steps and failures can be detected in the following way. First, a process waits for a certain reply from another process, for a given period of time. If the timeout expires and no reply is received, the process is considered to be crashed.

One of the fundamental results in distributed computing is also a negative one. The *Impossibility of Asynchronous Distributed Consensus with a Single Faulty Process*, also known in the literature as *the FLP-Impossibility Result* [27], proves that in the context of a purely asynchronous system, it is impossible to build a deterministic algorithm for agreeing on a one-bit value, that can terminate in the presence of even a single crash failure.

The intuition behind this impossibility result is based on the observation that, in a purely asynchronous system, a slow process cannot be safely distinguished from a crashed one. However, this result does not state that consensus can never be reached, merely that under the model's assumptions, no algorithm can always reach consensus in bounded time. There exist algorithms, even under the asynchronous model, that can (sometimes) reach consensus. However, this "bad" input may be unlikely in practice. Circumventing this impossibility result has been one of the main challenges for researchers, along the years. Their works have proven that by introducing a minimal set of properties that are satisfied in an asynchronous system, a deterministic algorithm for solving consensus, can be designed. Circumventing the impossibility result usually translated into modifying either the system model or the problem definition in order to allow the problem to become solvable.

One of the first solution for circumventing impossibility consists in using randomization to design probabilistic algorithms, as the impossibility result applies to deterministic algorithms. More precisely, the solution consists in replacing one of the properties that define consensus by a property that is satisfied with a certain probability. The so-called *randomized algorithms* choose to modify the Termination property. The new definition states that every correct process eventually decides with probability 1. Randomized byzantine consensus algorithms have been proposed in article such as [7, 53] and they rely on a random operation ("flipping a coin"), that returns either 0 or 1 with equal probability.

The notion of partial synchrony was introduced in [26]. A partial synchrony model makes the assumption that systems may behave asynchronously for some time, but they eventually stabilize and start behaving more synchronously. Algorithms based on this type of synchrony model make assumptions about time properties that are eventually satisfied. Termination is ensured only when these time properties are satisfied.

Algorithms that require certain constraints on initial values, represent another way of weakening the consensus definition and thus, circumventing impossibility. These algorithms terminate if the initial values satisfy certain conditions but satisfy the safety properties even if the conditions are not fulfilled [46, 47].

Apart from introducing stronger synchrony assumptions, another solution for circumventing impossibility is to consider a more general definition of the consensus problem. The *k-Set agreement problem* was introduced in [19] and it basically represents an extension of the classical consensus problem. Participants to the k-Set agreement problem are allowed to decide on different values. The number of different decision values is bounded by k. The Validity and Termination properties have the same specification as for the classical consensus problem. The Agreement property is more general than its corespondent in the basic consensus specification. In the particular case of $k = 1$, only one decision value is allowed. This unanimous decision leads to the classical consensus problem. Whereas the consensus problem has no deterministic solution as soon as at least one crash may occur, the k-Set agreement problem has rather straightforward solutions for $k > f$ (f being the maximum number of crash failures in the system). This result has been obtained in [19]. In [11], the authors show that the k-Set agreement problem is impossible to solve for any $k \leq f$.

Another major work-around consists in extending the asynchronous system with some synchrony properties, usually exploited by failure detectors.

2.3 Unreliable failure detectors

The *failure detector* concept was introduced by Chandra and Toueg in [16] with the purpose of identifying the minimal extensions to an asynchronous system that would render consensus solvable.

In [16], the authors identify two completeness properties and four accuracy properties. This classification leads to eight classes of failure detectors, which can be reduced to just four distinct classes, by applying reduction algorithms.

In the following, we provide some basic definitions for the two completeness and four accuracy properties. The *weak completeness* property requires that there is a time after which every process that crashes is permanently suspected by some correct process. If we strengthen

the constraint and we require that eventually every crashed process is permanently suspected by every correct process, we obtain the so-called *strong completeness* property.

The *perpetual accuracy* property requires that accuracy is always satisfied. *Eventual accuracy* relaxes this constraint: accuracy must be permanently satisfied only after some time. *Strong accuracy* states that no process is suspected before it crashes. The less constraint-full version of strong accuracy is *weak accuracy*: some correct process is never suspected.

Among the classes of failure detectors, we particularly focus on three of them, which are relevant for our work, namely *Eventually Weak*, denoted by $\Diamond W$, *Perfect*, denoted by P, and *Eventually Strong*, denoted by $\Diamond S$.

The *Eventually Weak* class of failure detectors is described in [16] as the weakest class that renders consensus solvable. A failure detector is in $\Diamond W$ if it satisfies weak completeness and eventual weak accuracy. These two properties require that eventually some conditions are satisfied and hold forever. However, in a real system in which properties cannot be satisfied forever, it is required that they hold for a sufficiently long period of time. If we refer to the consensus problem, we can quantify the "sufficiently long" period of time as being long enough to allow the algorithm to terminate and each process to decide.

The *Perfect* class, denoted by P, contains the failure detectors that satisfy the most constraint-full requirements, which are strong completeness and strong accuracy.

Another important class of failure detectors, intensively investigated in the literature, is the class of *Eventually Strong* failure detectors, denoted by $\Diamond S$. A failure detector belonging to this class satisfies the properties of strong completeness and eventual weak accuracy. This type of failure detector can make an arbitrary number of mistakes. In the context of an asynchronous distributed system, extended with failure detectors of class $\Diamond S$, several protocols for solving consensus have been proposed. In the following section, we describe the main principles of protocols that rely on $\Diamond S$ failure detectors.

2.4 $\Diamond S$ based protocols

2.4.1 Fundamental principles

The execution of a $\Diamond S$ based protocol usually proceeds in a sequence of consecutive asynchronous rounds. The concept of *round* is well suited for structuring fault tolerant asynchronous algorithms. Each round is uniquely identified by a round number r and the sequence of rounds is totally ordered. While executing a round, a process adopts the following behavior: it exchanges messages with other processes and, based on the gathered information, it performs several execution steps before proceeding to the next round. From the point of view of the communication, the round model is communication-closed: a process p_i communicates only with processes that are currently executing the same round as p_i.

The main purpose of the sequence of rounds is to ensure that processes will eventually converge towards a single value and decide on it. As rounds are executed asynchronously, different processes may be executing different rounds. Furthermore, they may decide during different rounds, due to crashes and failure suspicion patterns. In order to ensure that Agreement is always satisfied, protocols must make certain that all processes decide on the same value, even if they do not decide in the same round. This is usually achieved in the following way: each process maintains its current estimation of the decision value, which is

initially set to the process initial value. This value is updated during the protocol's execution and converges towards the decision value. Each process also maintains the current round number it is executing and includes this number in all messages it sends. Such a mechanism is called a *timestamp* or *tag* mechanism. Based on the round number (included as a field in the message), a process can filter the received information. By comparing the round number of the message with the round number of the process, messages received late are discarded while messages received early are buffered for future delivery.

Many of the $\Diamond S$ protocols are based on the *the rotating coordinator paradigm*. This concept assumes that any round r is coordinated by a process p_c, called the *coordinator of round r*. To ensure that only one process is the coordinator of a round r, the identifier of a coordinator p_c is computed according to the rule: $c = (r \ mod \ n) + 1$. The role of a coordinator is to determine and try to impose a decision value. If its attempt is successful during a round and if it is not suspected by any correct process, then the coordinator can safely decide a value and all other correct processes can safely agree on it.

With failure detectors of class $\Diamond S$, all processes may be added to the list of suspected processes. As they also satisfy the property of eventually weak accuracy, there is a correct process and a time after which that process is not suspected of being crashed. This important property ensures that eventually, a current coordinator will not be suspected by all other processes and will allow the algorithm to reach a decision value, thus ensuring the Termination property.

Another important assumption these protocols require, is that a majority of the processes in the system are correct, $f < \lceil \frac{n+1}{2} \rceil$, where n is the total number of processes in the system and f is the total number of crash failures. This assumption is essential for ensuring that Agreement is always satisfied. As processes may not decide in the same round, the protocol relies on majority sets to ensure the following property: if a process p_i decides a value v during a round r, no other decision value is possible for any other process p_j that decides in a round $r' > r$. These sets are also called *majority quorums*. The cardinality of a majority set is equal to $\lceil \frac{n+1}{2} \rceil$, where n is the total number of processes in the system. The main property of any two majority sets is that they always have a non-empty intersection. The majority requirement is also mandatory at certain steps during the computation: processes wait for messages from a majority quorum, in order to progress. As a majority quorum of processes is correct, the processes will not remain blocked while waiting for conditions that may not become true. The progress of the computation relies on this assumption.

The usual metrics for assessing the performance of a consensus protocol are the following:

- *the latency degree*: the minimum number of communication steps required by the protocol to reach a decision value.
- *the number of messages*: the minimum number of messages exchanged during the computation, between processes, before converging to a decision.

$\Diamond S$ based protocols also include a reliable broadcast mechanism for the decision value. Once a correct process decides, it will no longer participate in the protocol's execution and a process that has not decided yet may not learn the decision value. The main purpose of this mechanism is to broadcast the decision message to all other processes, such that if a correct process decides a value, then eventually every correct process also decides.

The following subsections provide a brief overview of some $\Diamond S$ based protocols, relevant for our work.

2.4.2 The Chandra-Toueg protocol

Description
Among the numerous contributions of [16], the authors also propose the first protocol that solves consensus by using a $\Diamond S$ failure detector, namely the *Chandra-Toueg protocol* (also abbreviated as the *CT* algorithm). In this approach, all messages are directed to and sent from the current coordinator. Therefore, such approaches are called *centralized*.

An execution of the *CT* protocol proceeds in asynchronous rounds, where each round is composed of four consecutive phases. As previously specified, each process p_i maintains an estimation of the decision value, est_i, set to the p_i's initial value, at the beginning of the computation.

The structure of a round r is described in Figure 2.3 and explained in the following:

1. **Phase 1:** Each process p_i sends its est_i value to p_c, the coordinator of round r. Along with this value, p_i also provides the so-called *timestamp* of the value, specified by the most recent round number in which p_i has updated its estimate. Initially, the timestamp is set to 0.

2. **Phase 2:** Coordinator p_c of round r gathers the messages sent during Phase 1 and selects a value from the received ones. The selection criterion is based on the highest timestamp observed by the coordinator in the received messages. In order to select a value, the coordinator must gather a majority of estimation values from processes. The cardinality of a majority set of replies is equal to $\lceil \frac{n+1}{2} \rceil$. After selecting the most recent value denoted by est_c, the coordinator sends it to all processes as the new estimation value.

3. **Phase 3:** When a process p_i waits for the est_c value from the coordinator, it must prepare a reply to send back to the coordinator. In order to do that, p_i queries its local failure detector. If the information provided by the failure detector places p_c among the suspected processes, then p_i will send a negative reply (a *nack*) to the current coordinator. Otherwise, p_i receives the message of the coordinator, updates its estimate to est_c, the timestamp to the current round number r and sends a positive reply (an *ack*) to the coordinator.

4. **Phase 4:** Coordinator p_c waits for replies from a majority quorum consisting of at least $\lceil \frac{n+1}{2} \rceil$ processes. If the set of gathered replies contains only positive feedbacks, a majority of processes have set their estimates to the same value est_c, in other words, the value est_c has been *locked*. Only a value that has been previously locked can become a decision value. The coordinator reliably broadcasts a request to decide the value est_c to all other processes. If a process p_i receives such a request, then it can safely decide est_c. If no value can be locked, in other words, the set of replies contains at least one *nack*, the coordinator proceeds to the next round.

As the system model assumes at least a majority of correct processes, this requirement ensures that a coordinator will never remain blocked during Phase 2 and 4, while waiting

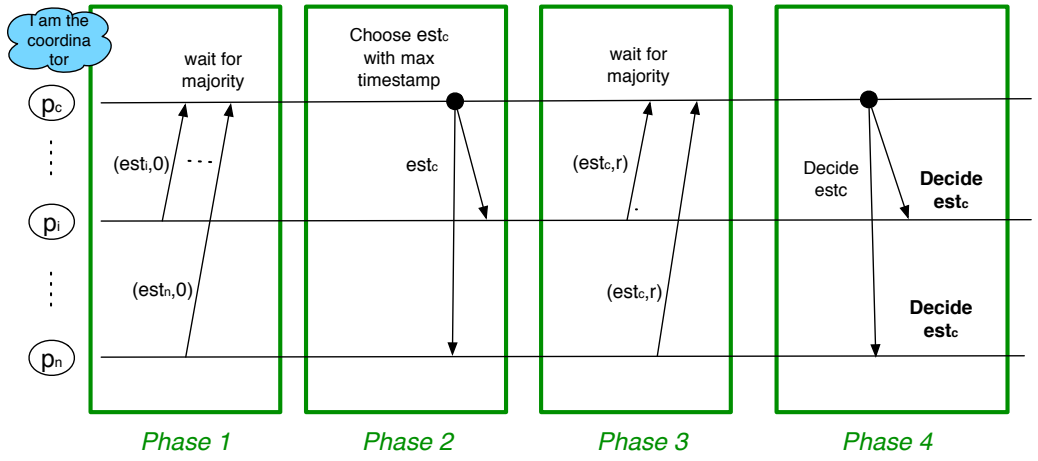


Figure 2.3: A best-case round in CT: p_c is correct and not suspected.

for a majority of replies from processes.

How exactly do majority sets ensure Agreement in the CT protocol?

Let us consider the execution of two consecutive rounds r and $r+1$. During Phase 4 of a round r, the coordinator decides a value v only after it has observed that a majority set of processes has adopted this value (value v has been locked). First, let us assume that a value v is selected during Phase 1 of the round $r+1$. During the first phase of the round $r+1$, the coordinator waits for a majority set of estimation values. As the value v has been locked during round r and as the intersection of two majority sets contains at least one process, the coordinator will observe this value v. The timestamp of this value is equal to r and it represents the highest timestamp possible, thus the coordinator will select this value as its new estimate. Therefore, the only value that can be locked and decided during round $r+1$ is v. If no value is selected during the first phase of round $r+1$ because the coordinator may crash, the same reasoning can be applied for other rounds.

Complexity Analysis

As described in [16], the *CT* protocol has a latency of four communication steps: in favorable circumstances, if the first coordinator is correct and not suspected by all other processes, each phase requires one communication delay. A trivial optimization allows to reduce this latency to three communication steps: round 1 can start directly with the second phase during which the coordinator broadcasts its estimate to all other processes.

Regarding the number of messages needed to reach a decision, we also refer to the best case scenario, in which the first coordinator is correct and able to decide in the first round. The total number of messages consists of the following: one estimate message sent by p_1 to all, $n-1$ replies from $p_2, p_3, ..., p_n$ to p_1 and one decide message sent by p_1 to all, which leads to $n+1$ messages sent during the computation, before reaching a decision. Therefore, the *CT* protocol requires $O(n)$ messages in the best case scenario.

2.4.3 The Early Consensus protocol

Description

Another protocol based on the rotating coordinator paradigm was proposed by Schiper in [54]. The *Early Consensus* protocol is based on the same main principles as the *CT* protocol. The main focus of this protocol is to improve the performance of algorithms based on the rotating coordinator paradigm and $\Diamond S$ failure detectors.

The protocol consists of two concurrent tasks: the first thread is triggered by the reception of a decision message. The main purpose of this task is to re-broadcast the decision message to all other processes, such that if a correct process decides a value, then eventually every correct process also decides. A second task executes the consensus protocol itself. Similar to the *CT* protocol, the execution of the *Early Consensus* protocol proceeds in asynchronous rounds, each round r_i being divided into two phases. In addition to the value itself, the estimation of a process p_i also includes the process identifier i of the process that proposed this estimation value.

The first phase of a round r attempts to decide on the estimation value of the coordinator. During this phase, the coordinator p_c of round r, broadcasts its value to all processes. When a process p_i receives this *estimation message*, it behaves as a relay: it passes on the received message to all other processes, after having adopted the value of the coordinator. In the best case scenario, if the coordinator p_c is not suspected, a process p_i can decide on the estimation value of the coordinator as soon as it observes this value in messages received from a majority set of processes. This favorable scenario is depicted in Figure 2.4. However, during the first phase, a process p_i can suspect the current coordinator, by relying on the information supplied by its $\Diamond S$ failure detector. In this case, p_i sends a *suspicion message*, through which it notifies all other processes that it suspects the coordinator. The transition from the first phase of round r to the second phase of the same round, is triggered by the following event: p_i gathers suspicion messages from a majority set of processes. During this phase transition, the estimate value of the coordinator is *locked*. As soon as p_i learns that a majority of processes have reached the second phase of round r, p_i can proceed to the first phase of round $r+1$. The main purpose of the second phase is to allow a faster convergence to a decision value, during the next round $r+1$: processes will start the next round with consistent estimate values. If a process has decided est_c during round r, then any process p_i that executes round $r+1$, will start with $est_i = est_c$. This *estimate-locking* was also applied in the *CT* protocol.

Complexity Analysis

In an optimal scenario, where the first coordinator p_1 is correct and no false suspicion occurs, the *Early Consensus* algorithm reduces the latency of reaching a decision to just two communication steps:

1. p_1 sends a message with is own estimate.
2. any process p_i receives and forwards this message.
3. any process p_i decides the estimate of the coordinator as soon as it gathers a majority set of messages that contain this estimate.

In the best case scenario, the pattern of messages issued by processes executing the *Early Consensus* algorithm is the following: p_1 sends its estimate message to all other processes,

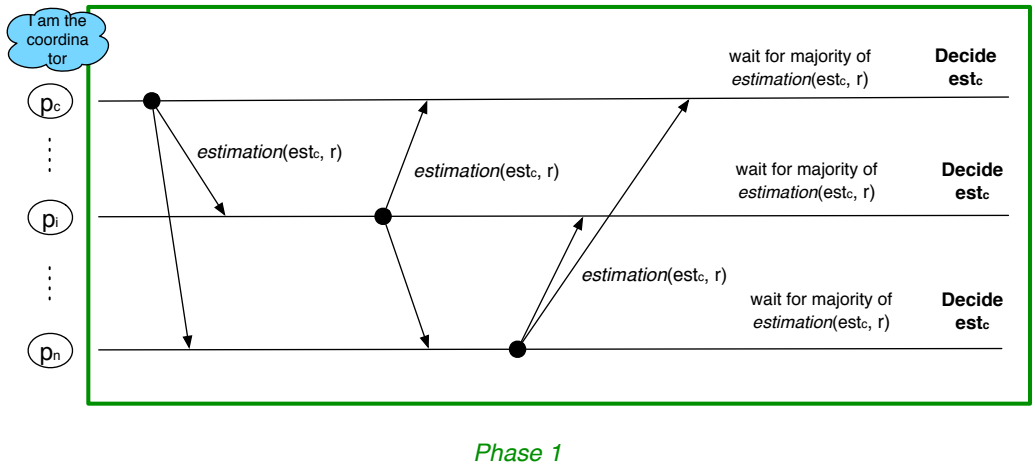


Figure 2.4: A best-case round in Early Consensus: p_c is correct and not suspected.

each process $p_2, p_3, ..., p_n$ re-transmits the estimate message to all other processes. The total of issued messages is N. Thus, the protocol requires $O(n)$ messages, the same as the *CT* protocol.

2.4.4 The General Consensus Protocol - A General Quorum-Based Approach

Description

The *General Consensus Protocol* described in [48], is designed upon the same concepts of asynchronous round execution and rotating coordinator paradigm. In addition to the two previously described protocols, this approach brings a novelty through its *generic* dimension: the protocol is designed such that it works with any failure detector of class S or $\Diamond S$. This generic dimension comes with an additional assumption. Let us define f as the number of processes that may crash and n as the total number of processes in the system. The protocol works with any failure detector of class S as long as $f < n$, whereas for any failure detector of class $\Diamond S$, it is required that $f < n/2$.

A round consists of two phases, each of which has a well-identified purpose. Similar to other $\Diamond S$ protocols, during the first phase, the coordinator of the round broadcasts its estimate value est_c. In addition to its own estimate value est_i, each process p_i also maintains a variable called $est\_from\_c_i$ that can take two possible values: the default value is a special mark denoted by $\perp$; or the value p_i has received from the coordinator. The first phase ends when either p_i has received the estimate value from the coordinator and updates $est\_from\_c_i$ to est_c, or when p_i suspects the coordinator (in this case, $est\_from\_c_i = \perp$). The purpose of the first phase is to ensure that at its end, each process p_i will have in its $est\_from\_c_i$ value either $\perp$ either est_c.

The purpose of the second phase is to ensure that Agreement is always satisfied. As processes may decide in different rounds, the second phase ensures the following: if a process p_i decides a value $v = est_c$ during a round r, then any process p_j that proceeds to round $r + 1$, will start the round with $est_j = v$. Each process p_i begins the execution of the second phase of a round r by broadcasting its $est\_from\_c_i$ value. The second phase is completed

only when p_i gathers est_from_c values from enough processes. "Enough" is quantified according to the failure detector class. The set of processes from which a process p_i receives est_from_c values defines *a quorum* and it is denoted by Q_i. Figure 2.5 describes an execution of the protocol in the best-case scenario in which p_1 is the correct coordinator and it is not suspected by any other correct process.

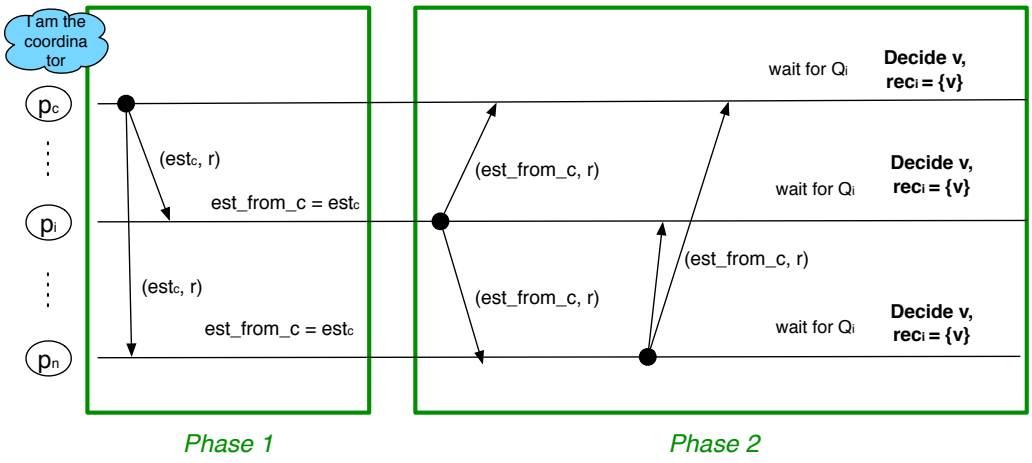


Figure 2.5: A best-case round in General Consensus: p_c is correct and not suspected.

To ensure that processes do not decide on different values, a constraint is imposed when defining quorums: *any two quorums Q_i and Q_j must have a non-empty intersection.* This is known as the *quorum requirement* and must hold for any pair of processes (p_i, p_j). Based on this requirement, a quorum is defined as a majority set of processes, in the case of $\Diamond S$ failure detectors. The quorum requirement states that there exists a process p_k, such that $p_k \in (Q_i \cap Q_j)$. This process has broadcast its $est\_from\_c_k$ value to both p_i and p_j, and furthermore, both processes have received its value.

A quorum is dynamically built during each round. Let rec_i be the set of values a process p_i receives from the processes belonging to Q_i. There are three possible cases for the values contained in rec_i :

1. $rec_i = \{\perp\}$
2. $rec_i = \{v\}$
3. $rec_i = \{v, \perp\}$

If a process gathers only values equal to v in his rec_i set, then it knows that all non-crashed processes have adopted v. Thus, p_i can safely decide the value v. If the rec_i set contains only $\perp$ values, p_i knows that a majority of processes suspects the current coordinator. In this case, no process can decide and p_i proceeds to the next round. Finally, if p_i observes both the $\perp$ value and the value v, p_i updates its estimate est_i to v, so that if a process decides v during round r, p_i will proceed to the next round with $est_i = v$.

Complexity Analysis

Let us consider the case in which the underlying failure detector behaves reliably and the first coordinator p_1 is correct. The latency degree of the protocol consists of two communication steps: p_1 broadcasts its value, each process relays this value, and once it has gathered a majority of such messages, it is able to decide.

The total number of messages issued in the best case scenario, is given by: one message sent by p_1, and each process $p_2, p_3, ..., p_n$ broadcasts its estimate to all other processes. The total amount of messages is $O(n)$.

2.4.5 Particularities

The three protocols previously described are built upon two types of interaction schemes: *centralized* and *decentralized*. These schemes are specified by the way the processes interaction is modeled and by the tasks performed by the coordinator as opposed to other processes. Both the *Early Consensus* and the *Quorum-based* protocols rely on a decentralized approach. In these protocols, the coordinator of a round behaves as an *initiator*, by broadcasting its estimate. Apart from this particular role, the coordinator and the rest of the processes behave in a similar way. In this sense, the protocols provide a uniform manner of describing the behavior of any process in the system. This is no longer the case in the *CT* protocol. An asymmetry is created between the current coordinator of a round and other processes. The coordinator executes special tasks and has an important role in selecting the estimation value and in deciding a value. The coordinator determines the estimation value and also notifies all other processes when a decision was made. Furthermore, the communication pattern is centralized: all messages are directed to the coordinator.

This constraint is relaxed in decentralized approaches as processes cooperate in a distributed manner. Any process can decide a value without needing a special notification from the coordinator. However, the condition for reaching a decision is similar: a process (coordinator or not) must learn that a majority of the processes have adopted the estimation value proposed by the coordinator. Another similarity comes from the fact that a process cannot participate in a previous round. Once it was completed, a round cannot be executed again.

In the *CT* protocol, the rule for progressing to the next round is the following: a process has sent a reply to the current coordinator, either a positive or a negative one, in case of suspecting the coordinator. The *Early Consensus* protocol introduces a *barrier* at the end of each round: a process must receive messages from a majority set before proceeding to the next round.

2.5 Other approaches to specifying the system model

The failure detector model augments the asynchronous system with external entities that provide useful information to the processes that query them. Contrary to this concept, there are several approaches that integrate within the system model all aspects related to synchrony and failures. These concepts are integrated within the system and they define the so-called *environment*.

GIRAF-General Round-based Algorithm Framework

A different approach for modeling round-based algorithms is introduced in [36]. The *General Round-based Algorithm Framework* allows to structure a round-based protocol into two separate components: the algorithm's computation during each round (controlled by the algorithm itself) and the round progress condition. The later is determined by an environment that satisfies specific failure detector properties. The framework defines *end-of-round* actions that are executed independently by each process. During each round, a process sends a message to all other processes and receives messages on incoming links, until the end-of-round actions are triggered. At this point, the oracle is queried and the message for the next round is computed. The environment encapsulates both the synchrony properties of the system and the failure model.

The HO Model

The *Heard-Of (HO) Model*, described in [17], provides a round-based model for structuring consensus protocols. In the HO model, the synchrony properties and the failure model are abstracted by the same concept: transmission failures are only detected without focusing on the faulty entity. Computation proceeds into rounds, similar to any other round-based model, with the particularity that any missed communication is lost. In each round, a process sends a message to all others and then waits to receive messages from the other processes. The model defines for each process p_i, the *heard-of set* as the set of processes from which p_i receives a message. The system properties are abstracted in a predicate defined over the collection of the *heard-of sets* obtained during a round.

2.6 Conclusions

In round-based protocols, computation proceeds in a sequence of asynchronous rounds. The usual behavior of a process during a round is the following: a process sends a message to other processes, waits to receive messages from some processes and then executes local computation to update its state. The purpose of executing rounds is to converge towards a single value and eventually decide on it. A round is a communication-closed layer: during a round, processes consume only messages sent during that round.

The rotating coordinator paradigm gives a special privilege to a single process during each round. This process is called the coordinator of the round and its main goal is to determine and try to impose a decision value. Regarding the interaction scheme between processes, centralized and decentralized approaches have been proposed. In decentralized models, the coordinator has the role of initiating a round by broadcasting its own value. After this first phase, all processes behave in the same way and each of them is able to determine a decision value and decide on it. In centralized schemes, the coordinator has the special task of deciding a value, any other process is able to decide if and only if it receives a decision message from the coordinator.

Chapter 3

The Paxos protocol

Contents

In the previous chapter we described failure detector oracles and we focused on $\Diamond S$ based protocols for solving consensus in asynchronous systems. This chapter addresses another type of oracles, namely *leader* oracles and focuses mainly on the *Paxos* protocol. After a brief overview of leader oracles, we then describe the main principles of the Paxos approach. We also focus on two major optimizations proposed in the literature for improving the performance of the protocol.

3.1 Leader Election Oracles

Oracles usually provide hints about the current status of the processes in the system. A *leader election* oracle represents a particular type of failure detectors. Whereas a failure detector provides a list of suspected processes, a leader election oracle supplies the identity of a process trusted to be correct. In [15], the authors introduce the weakest failure detector with which consensus can be solved. This failure detector is called Ω. Informally, an Ω oracle allows processes to eventually elect a common leader. Each process in the system queries its local Ω module and obtains the identity of the *current leader*, a process trusted to be correct at that time. Ω oracle may be unreliable as they may provide misleading information. This leads to different processes having different leaders. However, Ω guarantees that there is a time after which all processes have the same correct leader. In other words, Ω ensures that eventually a unique leader persists until the end of the execution.

The practical interest for the Ω oracle comes from the fact that many consensus algorithms, including Paxos, may be implemented by using Ω as a failure detector. However, Paxos does not require the assumption that eventually a leader is unique and persists until the end of the execution. Paxos succeeds as soon as minimal synchrony properties are satisfied for a sufficiently long period of time. Further details are provided in the following subsections. In [5], the authors address the problem of implementing Ω by considering two main aspects: the feasibility and the cost of an implementation.

3.2 History of Paxos

In this section, we briefly discuss the articles that introduced Paxos in the world of distributed consensus protocols.

3.2.1 The original Paxos

Paxos is a fault-tolerant protocol for solving Consensus in an asynchronous distributed system prone to crash failures. The original Paxos protocol was introduced by Leslie Lamport in [37] and revised later in [38].

Lamport has chosen to introduce the Paxos protocol in an unconventional manner: the name "Paxos" is derived from the greek island of *Paxos*. The inhabitants of this island formed the so-called *part-time parliament* that managed the legislative aspects of the Paxon society. The part-time parliament relied upon a consensus protocol. The Paxon legislators (members of the parliament) maintained consistent copies of the parliamentary records and communicate by messengers that delivered messages between legislators. Both messengers and legislators could leave the chamber of the parliament.

The data (ledger) is replicated at n processes (legislators). Operations (decrees) should be invoked (recorded) at each replica (ledger) in the same order. Processes (legislators) can fail (leave the parliament) at any time. At least a majority of processes (legislators) must be correct (present in the parliament) in order to make progress (pass decrees). The main purpose of the legislators is to issue law decrees concerning different aspects of the greek society. Each legislator maintains a *ledger* that contained the numbered sequence of the decrees that were passed. The main requirement for the ledgers is that of consistency: for a given decree

number, no two ledgers are allowed to contain different decrees. A decree would be passed and noted on the ledgers if there is a majority of the legislators that stayed in the chamber for a sufficiently long period of time.

3.2.2 Paxos made simple

« *The Paxos algorithm, when presented in plain English, is very simple.* »

Leslie Lamport

The part-time parliament protocol is also known in the literature as *the synod protocol*. It provides an efficient way of implementing a consensus service by replicating it over a system of (non-malicious) processes communicating through message passing. However, the simplicity and efficiency of the Paxos protocol failed to emerge from the metaphoric and rather complex approach in which the protocol was presented.

As the first attempt at introducing the Paxos protocol has proven to be difficult to understand, Lamport has later revised the Paxos solution in [38]. As the author himself states, this new attempt is much more clear and simple, as it "contains no formula more complicated than $n_1 > n_2$" [4]. In the following subsections, we provide a description of the Paxos protocol based on the version presented in [39].

3.3 Formal model

The Paxos protocol assumes an asynchronous distributed system in which processes communicate by message passing. The failure model is non-byzantine, which means that processes may fail only by crashing. The system model also assumes eventually reliable links; there is a time after which every message sent by a correct process to another correct process eventually arrives (*i.e.* there are no message losses).

3.4 The roles

The classical specification of the consensus problem requires that each participant to the consensus provides an initial value and then waits for the decision value. This is no longer the case in Paxos-like protocols. Indeed, a participant to a consensus instance is neither required to propose an initial value nor to wait for a returned decision value. In fact, a splitting into several roles allows to get free from the classical rigid interaction scheme. Regarding the structure of the Paxos-like protocols, Lamport has identified four basic roles [39]:

1. proposer (denoted herein by P_i)
2. learner (L_i)
3. coordinator (C_i)
4. acceptor (A_i)

Each process may take on a single or multiple roles. If f is the maximal number of failures that may occur, at least $f + 1$ coordinators, $f + 1$ learners, and $2f + 1$ acceptors should be defined. Herein, we assume that the protocol is executed by n processes with $f < n/2$. Proposers and learners are not counted.

Proposers are external entities that may provide initial values during a consensus instance. In an open system, their number and identities may vary in time. It is assumed that at least one non-crashed proposer supplies an input during each consensus instance.

Learners are in charge of detecting that the protocol has successfully converged toward a decision value. Proposers and learners are not involved in the convergence procedure, which is driven only by the interactions between coordinators and acceptors.

Coordinators and acceptors play a central role in ensuring that eventually a single value is selected to become the decision value. A *coordinator* is an active entity. When it is granted special privileges, a coordinator is in charge of selecting a value and trying to impose it as a decision value. An asymmetry is created between the coordinators by using a leader election service.

Paxos is a *leader-based* protocol. It relies upon a *leader election* mechanism, supplied by a *leader oracle*. *A leader oracle* outputs a unique process trusted to be correct, this process is called *the current leader* (see Section 3.1).

A leader election service is used to eventually grant a privilege to a single coordinator. Any coordinator can assume the role of the current leader. A coordinator determines if it should act as a leader by relying (directly or indirectly) on the information supplied by a leader election service[1]. The use of such a service ensures that a new process is selected as leader when the current leader is suspected to have crashed. The Paxos protocol is indulgent: it has been designed to never violate safety even if, at the same time multiple coordinators consider themselves leaders. In an asynchronous distributed system prone to crash failures, such a situation is always possible.

If a correct coordinator becomes the unique leader forever (or at least, till the current consensus instance ends), it is able to impose a selected value to a majority of acceptors and to detect the successful termination of its attempt. This extra synchrony assumptions (exploited by the leader election service) are required to circumvent the FLP impossibility result [27] and consequently to ensure liveness in the protocol.

Acceptors are passive objects that can accept or refuse to store a value suggested by a coordinator. Acceptors are used to implement quorums as majority sets. Therefore, by assumption, a majority of acceptors should never crash during the computation. The concept of *majority quorum* is detailed in Section 3.6.

3.5 The structure of the algorithm

3.5.1 Rounds

As described in [38], the Paxos protocol is concerned with solving a single consensus instance. The protocol's execution proceeds in a sequence of rounds. *A round*, also called *a*

[1] This service can be provided by a failure detector oracle Ω.

ballot [37], is identified by a round number denoted by r. Round numbers must fulfill the following properties (among others):

- a round number is unique and locally monotonically increasing.
- the number of rounds required to solve a consensus instance is unbounded.
- a relation of total order is defined among the round numbers (also called the *ballot numbers*).

When a coordinator is elected as a leader, it must determine the round number (or the ballot number) under which it will execute an attempt to converge towards a decision. To guarantee that two leaders will not choose the same round number, this monotonically increasing value reflects the identity of a single coordinator. For example, each coordinator C_i can take an initial value equal to i for its first attempt and later increase it by n (or by any multiple of n) at each new attempt. In this way, a round numbered r is coordinated by a single coordinator whose identity is $r \mod n$.

The Paxos protocol is based on a *timestamp* mechanism which is defined by the round number. All communication messages between acceptors and coordinators are timestamped with the round number. Each acceptor keeps track of the highest round number ever observed. An acceptor responds only to the leader with the highest round number seen so far and discards all other requests with lower round numbers.

During each round it participates in, an acceptor casts a vote to adopt only one value proposed during that round by the current leader. As several leaders can potentially act at the same time, the round numbers allow to distinguish between values proposed by different leaders. An acceptor agrees to participate only to the round that has the highest timestamp (round number).

3.5.2 Two phases of Paxos

The execution of a round proceeds in two phases: a *Prepare* phase during which the leader gathers information from acceptors, and a *Propose* phase that represents the attempt of the leader to impose a value. If this attempt is successful, the leader can safely decide the value. Both phases are initiated by the current leader and the *Propose* phase is always executed after a *Prepare* phase. In each of the two phases, the leader contacts the acceptors and waits for replies from a majority of them.

In Paxos-like protocols, only a leader can interact with acceptors. This interaction is achieved by broadcasting either a *Read* request or a *Write* request. In the Paxos terminology, a *Read* request is sent during the *Prepare* phase, while a *Write* request is sent during the *Propose* phase. A *Read* request carries only a round number, while a *Write* request contains both a round number and an attached value.

An execution of the Paxos protocol in well behaved runs (when no crashes occur) is depicted in Figure 3.1. The figure describes the behavior of coordinators, acceptors and learners (coordinator C_1 acts as the current leader).

Prepare phase

The main role of the *Prepare* phase is to ensure consistency with actions performed by

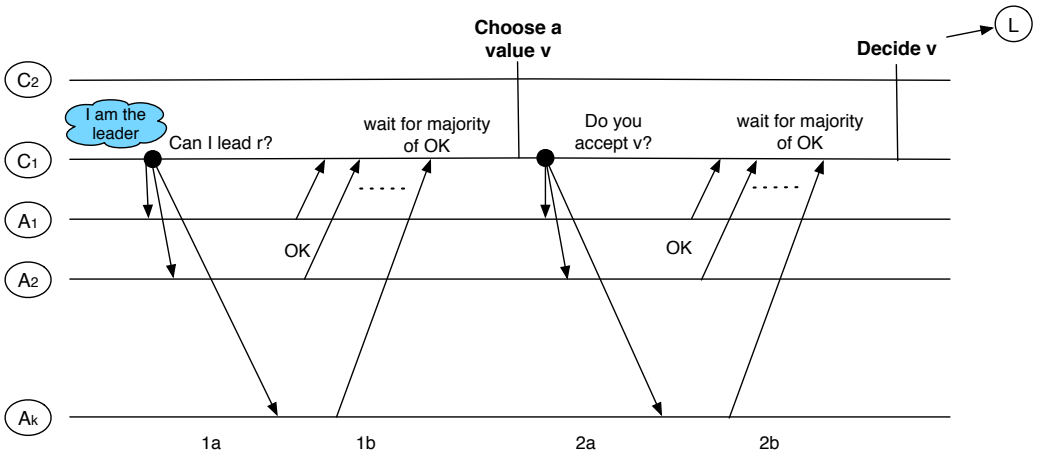


Figure 3.1: Paxos in well behaved runs.

previous leaders. During this phase, a leader communicates its new round number to the acceptors (*Read* operations). If it gathers enough feedbacks from them, it can switch to the *Propose* phase. The leader contacts the acceptors, asks them to join his round and obtains information about possible past decisions.

The *Prepare* phase is always performed when a new round is initiated. Any coordinator can query its leader election module at any time. If it believes it can act as the current leader, any coordinator can initiate a new round, more precisely a *Prepare* phase. A leader starts the *Prepare* phase by broadcasting a *Read* request carrying a round number (previously selected by the leader, as explained above). This step is also called *Phase 1a*. As each acceptor keeps track of the highest round number ever observed, the round number contained in any request sent by a leader is used by an acceptor to discard old requests. A reply returned by an acceptor contains the last value adopted by this acceptor as well as the round number during which this last update has been done. This reply is also called a *Phase 1b* message and it is depicted in the figures by the *ok* message. This messages contains three fields of information: the round number, the last value adopted by the acceptor and the round number during which it adopted this value. By agreeing to participate to a round r, an acceptor also makes the implicit promise not to participate to any other round $r' < r$. However, it may still take part in a round $r' > r$. An acceptor's reply itself represents a positive acknowledgment. If an acceptor does not agree to participate to a round, it will just ignore the request without sending a negative reply. Figure 3.2 zooms on the *Prepare* phase.

Phase Transition

If a leader gathers enough positive replies during a *Prepare* phase of round r, it switches to the *Propose* phase r. The notion of "enough" is quantified by *majority quorums*. A leader can proceed to the *Propose* phase only if it has received replies from a majority quorum of acceptors. Otherwise, the leader will execute a new *Prepare* phase with a higher round number. The concept of *majority quorums* is essential for ensuring safety and will be detailed in Section 3.6. When this phase transition occurs, the leader uses

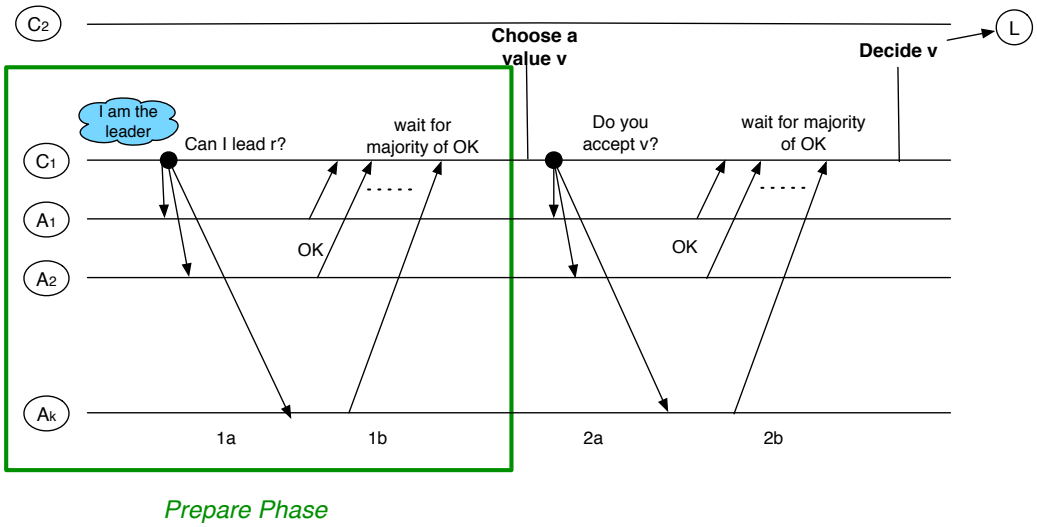


Figure 3.2: Paxos: Prepare Phase.

the information collected during the *Prepare* phase to determine the origin of the value it will use during the *Propose* phase. Two cases are envisioned. If at least one acceptor has informed the leader of a value previously proposed by another coordinator acting also as leader during the same consensus instance, the leader must select one of the most recent values among such values. The most recent value is the value adopted by acceptors during the highest round number ever observed. The selection of such a value is mandatory to ensure safety. Otherwise, if no such value exists, any value can be chosen by the current leader. In this case, the chosen value can be provided by a proposer to the current leader.

Propose Phase

Any leader executing the *Propose* phase of a round has previously executed the corresponding *Prepare* phase. During the following *Propose* phase, a *Write* operation executed by the leader aims at suggesting a safe value to the acceptors. If enough acceptors follow this suggestion, this value becomes the decision value. The leader broadcasts a *Write* request (also called a *Phase 2a* request) to all acceptors. This request carries the previously selected value and the round number. Once it receives such a message, an acceptor adopts the value of the leader only if it still takes part to the round number contained in the request or to a lower round number. If another leader has acted with a higher round number, the acceptor has proceeded to another round and will ignore the request of the leader. In order to complete the *Propose* phase, a leader must gather replies from a majority quorum of acceptors. These replies, also called *Phase 2b* messages, inform the leader that a majority of acceptors have adopted the value suggested by the leader. In this case, the leader can safely decide the value.

Figure 3.3 emphasizes the steps of the *Propose* phase.

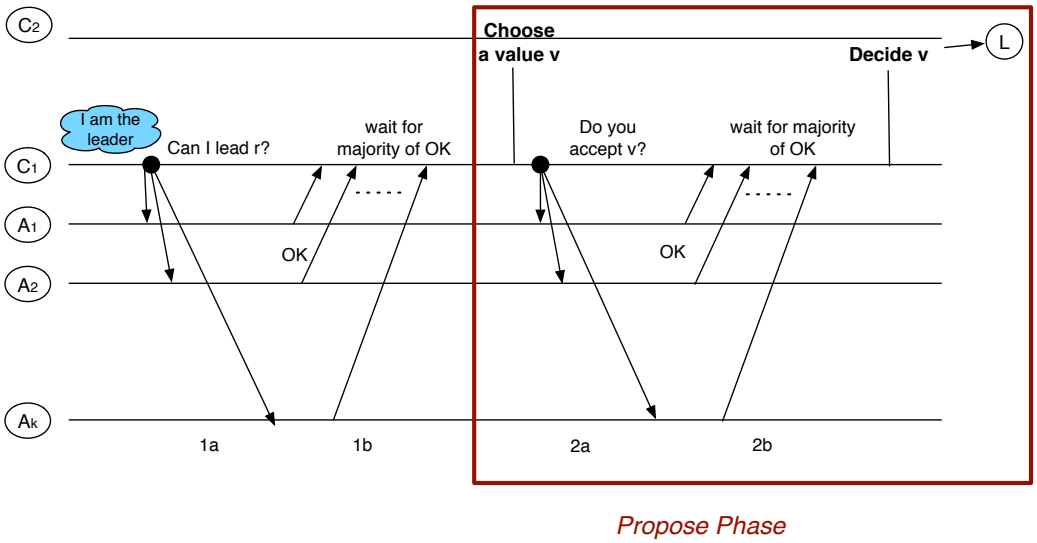


Figure 3.3: Paxos: Propose Phase.

3.6 Main principles - why does Paxos work?

Paxos is an efficient yet simple protocol as it relies on some principles that ensure that Paxos always works.

Never go back. Each participant to the Paxos protocol adopts the same behavior: it participates only to rounds higher-numbered than the previous round. An acceptor always joins the newest round it learns about and it will even abandon old rounds if necessary.

Paxos is guaranteed safe. To ensure safety, all Paxos-like protocols rely on the concept of **Majority Quorum**. In [39], Lamport defines quorums as majority sets of acceptors: a quorum represents a majority of correct acceptors. The **Quorum Requirement** states that any two quorums must intersect. More formally, if Q and R are acceptor-quorums, then $Q \cap R \neq \emptyset$.

To decide a value v_1, a learner must detect that a majority of acceptors have sent a positive reply in response to *Write* operations sent by the same leader during the same *Propose* phase (*i.e.*, all these *Write* operations are equivalent and contain the same round number r_1 and value v_1). Later, to end a *Prepare* phase with a round number r_3 such that $r_3 > r_1$, a leader must also gather a majority of positive replies. As two majority quorums intersect, among the positive replies to the Read operation, at least one acceptor indicates that its current value v_2 has been obtained during round r_2 with $r_1 \leq r_2 < r_3$. A reasoning by induction on the value r_3 allows to conclude that necessarily $v_1 = v_2$.

Paxos is not guaranteed live. The protocol never blocks, however the termination is ensured once the leader election service stops making mistakes and provides a single identity of the leader for every process that invokes it. Once there is a correct leader, it chooses the highest round number. No other process becomes a leader with a higher round number. All correct process reply to its *Prepare* message, accept its proposed value and decide on it.

Once reached, a decision is stable. Once the protocol reaches consensus, this property is

never violated and the agreed value is never changed.

3.7 The latency of classic Paxos

The Paxos protocol introduced by Lamport in [37, 38] is also known in the literature as the *Classic Paxos*.

The usual metric for evaluating consensus protocols is the *latency*. The *latency of reaching a decision* is defined by the number of communication steps that link two events: "a value is available at a proposer" and "a decision value is acquired by a learner".

This communication path depends on many factors that influence the protocol's execution and varies according to different scenarios. In a best case scenario, each of the two phases of Paxos require two communication steps. The stability of the current leader influences the performance of the protocol, as a new leader must start a new round with a new *Prepare* phase. Another aspect that must be taken into account is the time at which the proposal arrives; more specifically, the computation step reached by the protocol when the value provided by the proposer becomes available.

During periods of asynchrony, the leader election service makes many mistakes and the leader changes often. In this case, the latency is finite but unbounded, as each new leader will start a new round by executing a *Prepare* phase, until one leader is stable for a long enough period.

During stable periods, in which the leader remains stable for sufficiently long time, a proposer is linked to a learner by a communication path of length six in the worst case (proposer $\rightarrow$ leader $\rightarrow$ acceptors $\rightarrow$ leader $\rightarrow$ acceptors $\rightarrow$ leader $\rightarrow$ learner).

The latency can be better if we consider that a coordinator can also behave as a learner. Let us also notice that, while the message sent by a proposer to a leader is in transit, the leader can perform simultaneously the first step of the *Prepare* phase. In this case, the replies of the acceptors for the *Prepare* phase and the two steps required by the *Propose* phase add three more message delays. Thus, the latency of classic Paxos consists of four communication steps.

In the best case scenario, if the value provided by a proposer becomes available after the *Prepare* phase is completed, the latency is reduced to three communication steps: one corresponding to the message sent by the proposer and two other message delays required by the *Propose* phase.

3.8 Making Paxos fast

3.8.1 Motivation

As consensus protocols are intensively used as building blocks for higher-level applications, many research works have been devoted to optimizing their performance. Many fault-tolerant techniques are implemented by relying on a unique and everlasting sequence of decisions. Such a sequence is usually constructed by invoking repeatedly a consensus service provided by a dedicated set of n nodes. As presented in [39], the description of Paxos focuses mainly on a single consensus instance. The Paxos protocol (and more generally, any

consensus protocol) can be used as a basic building block to solve a sequence of consensus instances. This can be achieved by launching separate instances of the consensus protocol.

3.8.2 State Machine Replication

Producing a unique and everlasting sequence of decisions is at the cornerstone of the *State machine approach* [55] which aims at creating a sequence of commands. In this problem, the external clients are the processes that are issuing commands. The servers are the processes in charge of executing those commands according to the unique total order defined by the core. Each client participates to some consensus instances (but not all). The *Propose* primitive accepts a single parameter, namely the proposed value. If a Proposer provides a value v_x then this proposed value will eventually appear in a future decision $<c , v_x>$ but the client ignores (and has no control on) the value of c when it submits v_x.

In this approach, data is replicated at n servers. This technique relies upon the client-server interaction: each process (client/server) has an estimate of who the current leader is. Clients issue operations that need to be performed in the same order at all correct servers. A client sends a request to the current leader that launches the Paxos consensus algorithm to agree upon the order of the requests. Once the consensus is completed, the leader sends the response back to the client.

To implement the state machine approach, Lamport suggests to tag each command with the round number (also called ballot number) that is used in the protocol. Yet, this strategy may lead to have holes in the sequence and, consequently, *nop* commands may have to correspond to some sequence numbers. In [40] and [43], a single everlasting instance of the protocol manages both a round number (like in Paxos) and a consensus number that are initialized once.

3.8.3 FastPaxos

Regarding the Paxos algorithm, two main strategies have already been proposed in a recent past, namely FastPaxos (without space) described in [8] and Fast Paxos (with a blank) presented in [39]. In the following subsections, we provide an overview of these two optimizations. We aim at describing the context that enables each optimization, their main functioning principle and the gain obtained for each of the two strategies.

A first strategy leads to a performance gain if circumstances are favorable when executing several consecutive consensus instances. The context that enables the use of this optimization consists in long lasting failure-free synchronous periods in which the elected leader remains stable. This optimization strategy tries to benefit from the stability of an elected leader. When a coordinator is the unique leader for several consecutive consensus instances, it does not have to run a *Prepare* phase (more precisely, *Read* operations) followed by a *Propose* phase (*Write* operations) for each new consensus instance. In fact, the *Prepare* phase is executed only when a coordinator becomes the current leader to ensure consistency with possible actions made by previous leaders. Once a *Propose* phase begins, it lasts as long as no other leader appears. Consequently, several consecutive consensus instances can be solved within the same *Propose* phase, as long as no other leader appears. Moreover, except the first one, each new consensus instance just requires four communication steps in favorable circumstances: proposer $\rightarrow$ leader $\rightarrow$ acceptors $\rightarrow$ leader $\rightarrow$ learner (depicted in Figure 3.4).

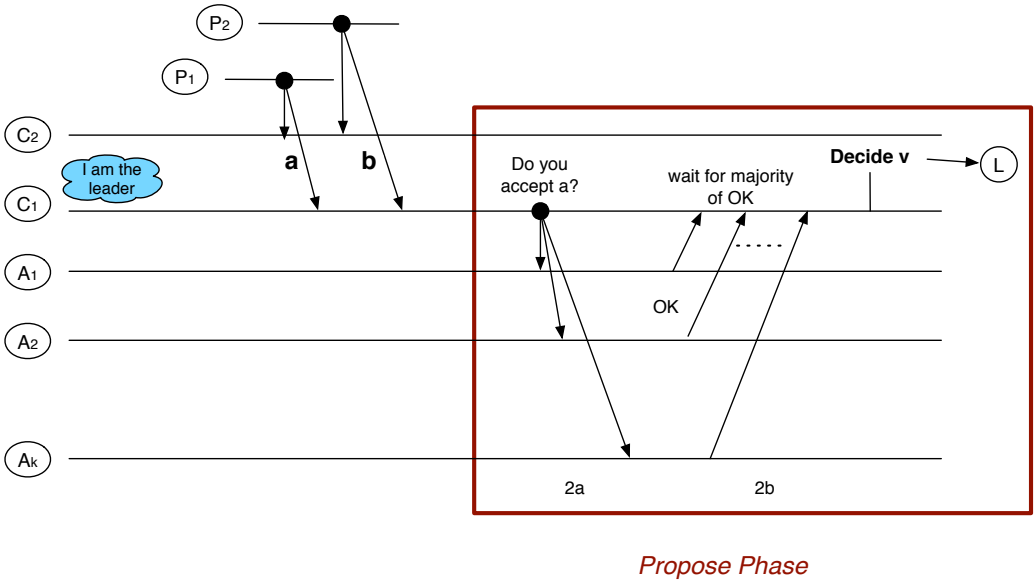


Figure 3.4: FastPaxos.

The main principle of this optimization consists in the removal of the *Prepare* phase. As long as the current leader remains stable, it will execute only *Propose* phases. A new *Prepare* phase must be run only when a leader change occurs. This optimization was already suggested in the original descriptions of the Paxos protocol [37, 38]. The FastPaxos protocol (without space) presented in [8] includes among numerous contributions a rather complex implementation of this optimization. The principle used to reduce the number of communication steps in FastPaxos is also adopted in other works: for example, by Lampson in [40] (where the notion of view is proposed) and in the work of Martin and Alvisi [43] (where the concept of regency is introduced).

In [9] (with a $\Diamond$S-based consensus protocol) and in [8] (with a Paxos-like protocol), the authors suggest to keep, even after the end of a consensus instance, the identity of the coordinator that has made the last decision and to reuse this information during the next consensus instance. In [8], when a (potentially new) leader starts the next consensus instance, this information can be exploited to optimize the decision latency. Indeed, if the leader has not changed in between, the first phase required in the original Paxos protocol (called the *Prepare* phase), is useless and in favorable circumstances, the new consensus instance just requires four communication steps.

3.8.4 Fast Paxos

Fast Paxos [39] involves two types of execution modes: a *classic* and a *fast* mode, that correspond to *classic* and respectively *fast* rounds. The *classic* mode functions just the same as *Classic* Paxos. In [39], there exists a static *a priori* agreement between the participants, regarding the round numbers that will be executed in the *fast* mode.

This second strategy, presented by Lamport in a protocol called Fast Paxos (with a blank) [39], tries to take advantage from a low throughput of the flow of initial values pro-

vided by the proposers. It aims at reducing the number of communication steps to three (proposer → acceptors → leader → learner), in favorable circumstances. If the most recent consensus instance has been completed for a long time, instead of being idle, the current leader can anticipate some part of the computation for the next consensus instance. The leader prepares the next consensus instance by sending a special value, called an *Any* value, to the acceptors. Once an acceptor receives it, it knows it is allowed to adopt a value directly provided by a proposer, which sends an initial value to both acceptors and coordinators. This is also called a *fast* round. In favorable circumstances, a gain can be obtained: if all proposers provide the same initial value, as such an initial value does not pass in transit through the leader and therefore the decision latency is reduced. These communication steps are described in Figure 3.5.

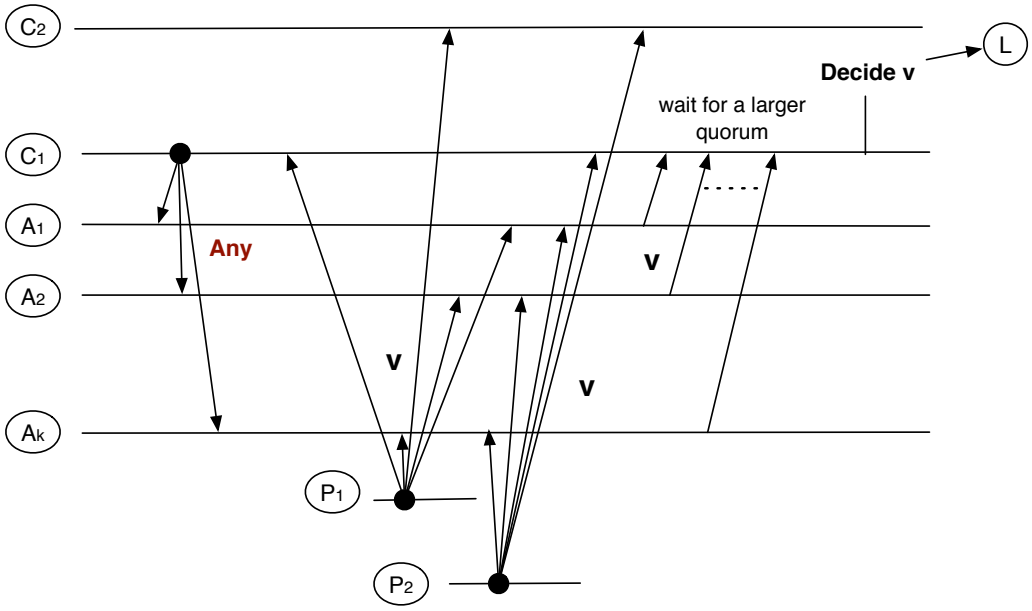


Figure 3.5: Fast Paxos.

Unlike Paxos, in Fast Paxos [39], the values adopted by the acceptors during a round r are not necessarily equal. Different proposers may provide different values. These proposal messages may be received in different orders by different acceptors. In such cases, a *collision* may occur. Consequently, a more restrictive definition of quorums has to be used to assess the successful termination of either a Read or a Write operation. As indicated by Lamport [9], majority quorums have to be replaced by larger quorums called herein *Any* quorums. As an *Any* quorum is larger, the maximal number of failures f that are tolerated has to be lower. Moreover, these larger quorums are more difficult to obtain as they require to collect more replies from the acceptors.

In [39], Lamport defines quorums as sets of processes (acceptors). Each round has a set of quorums associated with it. Classic rounds use classic quorums while fast rounds rely on fast quorums. These sets of acceptors must satisfy some given properties called *Quorum Requirements*. These properties state that (1) any two quorums must have a nonempty intersection and (2) any quorum and any two fast quorums from the same round must also

have a nonempty intersection. More formally, the *Fast Quorum Requirement* states that for any rounds i and j:

Let an *i-quorum* denote the quorum of acceptors used during round i.

- if Q is an *i-quorum* and R is a *j-quorum*, then $Q \cap R \neq \emptyset$.

- if Q is an *i-quorum*, R and S are *j-quorums* and j is a *fast* round, then

$Q \cap R \cap S \neq \emptyset$.

Based on these requirements, Lamport has defined in [39], the minimum number of acceptors that can constitute a quorum. Let us assume that N is the total number of acceptors that are in the system. The cardinality of any classic quorum, Q_c, and of any fast quorum, Q_a, can be computed as follows:

$$|Q_c| \geq \lfloor N/2 \rfloor + 1, \qquad |Q_a| \geq \lceil 3N/4 \rceil.$$

The same requirements for quorums cardinalities are also obtained in [60].

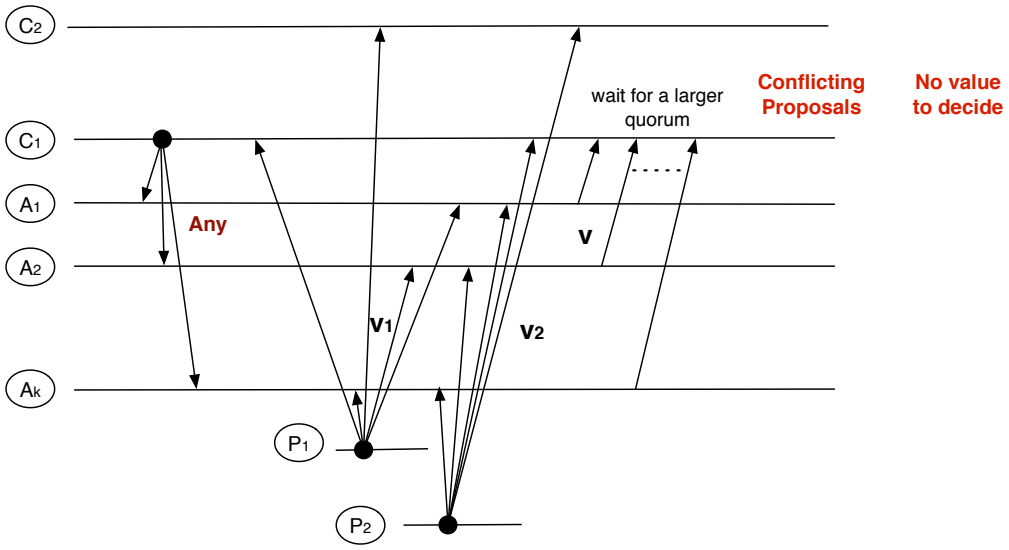


Figure 3.6: Collisions.

When different values are proposed simultaneously, this optimization can be counterproductive as it may require the execution of a time consuming recovery procedure. In case of competing proposals, no value can be safely chosen (see Figure 3.6). The usual way to recover from such a collision is to begin a new round. A coordinator C_i that learns of a collision in round i must start a new round with a number $j > i$. More precisely, C_i must initiate a *Prepare* phase by sending a *Read* request. In [39], Lamport suggests to optimize the classical mechanism of recovering from collisions. However, this optimization is only possible under stronger assumptions. If i is a fast round and C_i is coordinator of rounds i and $i+1$, the information last sent during round i can be used during round $i+1$. Based on this observation, C_i can skip the *Prepare* phase for round $i+1$ as it knows that no one else has acted as a leader between rounds i and $i+1$. Therefore, round $i+1$ can begin directly with the *Propose* phase. Two *collision recovery* mechanisms are described in [39], namely *coordinated* and *uncoordinated* recovery.

The main disadvantages of this optimization are the following:

- it relies on a static *a-priori* convention on the *fast rounds* numbers.
- larger quorums (*Any* quorums) must be used.
- if collisions are frequent, it can be counterproductive as it requires an expensive recovery procedure.

3.8.5 Related work

These two types of optimizations (*Fast Paxos* and *FastPaxos*) are also combined in works as [30] and [25]. In [30], Guerraoui and al. introduce the notion of*refined quorum system*: the optimization proposed by Fast Paxos, is a particular case of this more general model. They consider byzantine failures and propose a solution to the consensus problem that integrates the two optimizations in each round of a consensus instance. In an inappropriate scenario, where different proposers send different initial values, the use of optimization Fast Paxos prevents a decision during the initial round and delays the computation. In [25], Dobre and Al. propose a solution that relies also on these two optimizations. During each round, they simultaneously manage an execution with optimization FastPaxos only and an execution with both optimizations. Whatever the circumstances, they obtain a latency corresponding to the faster strategy. Yet the solution requires a few more messages and this hybrid solution is well suited to the case of wide area network where the cost of communication has a high impact on the time required to reach a decision. While optimization Fast Paxos is activated during each round in [30] and [25], there exists a static *a-priori* agreement between the participants, regarding the use of an *Any* value during a round in [39].

The main disadvantage of *Classic Paxos* is the high dependancy on the availability of the current round coordinator. In case of a leader failure, extra communication steps are required in order to resume normal execution: the failure of the leader must be detected, a new leader must be elected, the new leader has to synchronize with a quorum of acceptors etc.

Multicoordinated Paxos [13] introduces a third type of execution mode, the *multicoordinated* mode, besides the *fast* and *classic* modes. During a multicoordinated round, the role of a coordinator is played by a quorum of coordinators. Proposers send their proposals to the coordinator quorums of the current round, instead of a single leader. Each round i has a set of *coordinator-quorums* (a set of sets of coordinators), also denoted by *coordquorum*. An *i-coordquorum* represents a coordinator quorum employed during round i and it is defined as a majority of coordinators. Acceptors adopt a value only if it is forwarded by a quorum of coordinators. When consensus runs in the *classic* mode, a round has a single quorum of coordinators, containing a single element. The *Coordquorum Requirement* states that for any classic round i, if Q and R are *i-coordquorums*, then $Q \cap R \neq \emptyset$.

Collisions may also occur in multicoordinated rounds: if the quorums of coordinators receive different values from different proposers, the coordinators will forward to acceptors different values for the same round. In this case, acceptors will not be able to accept any value and the algorithm requires extra communication steps for recovery. However, a collision in *multicoordinated* mode are different than collisions in *fast* mode because in the latter, acceptors must write in stable storage every time they accept a value, while coordinators do not have to. The main advantage of *Multicoordinated Paxos* is the low dependency on the

availability of the current leader. Indeed, a single coordinator failure does not prevent values from being learned.

3.9 Conclusions

This chapter provides an overview of the Paxos protocol [37, 38] and several of its variants. The Paxos protocol introduced by Lamport in [37] represents an efficient and clever solution for solving consensus in an asynchronous system. After describing the underlying principles of the protocol, we focus on variants of Paxos, proposed with the purpose of optimizing the protocol's latency. The motivation of these protocols comes from the repeated use of consensus protocols as building blocks for higher-level application, among which the state machine approach represents a powerful example.

Regarding the Paxos protocol, two main optimizations have been proposed for reducing the latency of learning a decision: FastPaxos [8] and Fast Paxos [39]. From now on, we denote these two optimizations by $\mathbf{S_O}$ and respectively by $\mathbf{R_O}$. For each of the strategies, we provide a description that focuses on the following main aspects: the context that enables the optimization, the main principle of the optimization and the improvement in the protocol's latency. However, the optimization $\mathbf{R_O}$ succeeds only in favorable circumstances, otherwise it can be counterproductive and it increases the total latency of the protocol.

Part II

Contribution: Paxos-MIC, an Adaptive Fast Paxos

Chapter 4

Paxos-MIC - An Adaptive Fast Paxos

Contents

THIS chapter presents the algorithmic contribution of our work: the development of *Paxos-MIC* - an efficient framework for making quick everlasting decisions. We first define the *Multiple-Integrated Consensus* problem that allows us to identify the context of our approach and the motivation for proposing our protocol: the focus on both the construction and the availability of a sequence of decision values. We briefly discuss the system model and also the architecture of our protocol from the point of view of the entities involved and the communication scheme. The presentation of the Paxos-MIC protocol is done in two steps. We describe a first version of the protocol that only integrates optimization $\mathbf{S_O}$ (used in FastPaxos [8]). Then, we present the modifications required to use (depending on the context) both optimizations. This two-step description allows to clearly identify the parts of the protocol that are impacted by optimization $\mathbf{R_O}$ (introduced by Lamport in Fast Paxos [39]).

4.1 The Multiple-Integrated Consensus problem

In an asynchronous distributed system prone to crash failures and message losses, the problem of making a unique and everlasting sequence of decisions is crucial as it lies at the heart of many fault tolerant techniques. A well-identified subset of n nodes (called herein the *core*) is usually in charge of serving this never-ending need for agreement. On one hand, members of the core interact with external *proposers* (through the *Propose* primitive) and external *learners* (through the *Decide* primitive). On the other hand, they cooperate among themselves to establish the sequence of decisions. Each decision is a pair $< v_x, c >$ which can be delivered to any interested external learner. The integer c identifies the consensus instance during which a decision $< v_x, c >$ is reached while v_x denotes the corresponding decision value.

An external proposer participates to a consensus number c by invoking the *Propose* primitive with two parameters: the consensus number c and an initial value v. During the execution of a consensus instance c, a value v, chosen among the proposed initial values, becomes the unique decision value. The notation $< v, c >$ specifies this decision value. An external proposer that has sent a proposal during a consensus instance c is also regarded as an external learner to which the information $< v, c >$ must be transmitted.

If a protocol that solves the consensus problem [42] is available, separate occurrences of this protocol can be launched to construct step by step the sequence of decisions. The decision $< v_x, c >$ is the outcome of its c^{th} execution. Formally, the problem of building a sequence of decisions is defined by the following three properties.

1. **Validity**: If a process decides $< v, c >$, then an external proposer has previously called Propose(v,c).
2. **Agreement**: If a process decides $< v_x, c >$ and another process decides $< v_y, c >$, then $v_x = v_y$.
3. **Termination**: If a correct external proposer p calls infinitely often Propose(v_x, c) and if either $c = 1$ or a process decides $< v_y, c - 1 >$, then p eventually decides $< v_z, c >$.

By definition, during each consensus instance, the *safety* properties that characterize the consensus problem are satisfied. The Validity property states that the decision value v_x that appears in $< v_x, c >$ must be selected among the proposed values that are available during the consensus instance numbered c. Due to the Agreement property, if two learners obtain two decisions $< v_x, c >$ and $< v_y, c >$ then the equality $v_x = v_y$ necessarily holds. The Termination property indicates that any correct external proposer that provides infinitely often an initial value for the consensus instance number c, will eventually obtain a decision, once all the previous consensus instances (numbered from 1 up to $c - 1$) will be finished: at least one process has decided during each of them. Due to this last property, the problem we want to solve is slightly different from the consensus problem.

Producing a unique and everlasting sequence of decisions is at the cornerstone of the *state machine approach* [55] (detailed in Section 3.8.2) which aims at creating a sequence of commands. In this problem, the external proposers are the processes that are issuing commands. The external learners are the processes in charge of executing those commands according to the unique total order defined by the core. Each external proposer participates to some consensus instances (but not all) and it is not aware of the consensus instance. The *Propose* primitive accepts a single parameter, namely the proposed value. If a proposer provides

a value v_x then this proposed value will eventually appear in a future decision $< v_x, c >$ but the proposer ignores (and has no control on) the value of c when it submits v_x.

The *Repeated consensus* problem [23] requires also to solve a sequence of consensus instances. In this problem, each correct proposer originates an infinite sequence of proposed values. The *Propose* primitive accepts two parameters: when a proposer provides the proposed value v_x, it also indicates the consensus number c during which this value may become a decision value. By definition, this proposer has already participated to the $c-1$ previous consensus instances. In the Repeated Consensus problem, n proposers and n learners are considered and are supposed to be the n members of the core. Each consensus instance starts in an initial configuration [18] defined as a collection of n initial values (*i.e.*, one proposed value per node).

Within this work, we consider the *Multiple-Integrated Consensus* problem which is a mix of the above problems. As it was the case in the state machine approach described in Section 3.8.2, the number of external proposers is unbounded and none of them is supposed to participate to all the consensus instances. A *participant* to the consensus c is a node $PLext_k$ (correct or not) that invokes *Propose(v_k,c)*. By assumption, the number of participants to a consensus c is greater or equal to 1, but unbounded: there exists at least one correct participant per consensus. When a proposer provides a value to the core, it has to identify the consensus instance during which its proposed value may become a decision value. For the sake of simplicity we assume that each external proposer is de facto an external learner interested by the decision value. Herein, each of them is called an external proposer/learner and denoted $PLext_1$, ..., $PLext_k$, ... Each consensus instance satisfies the classical validity, uniform agreement, and termination properties. To ensure the liveness property, we assume that, for each consensus instance c, there exists at least one correct external proposer/learner that is able to propose an initial value.

In most practical settings, a consensus instance c starts only if the previous one (numbered $c-1$) is already completed (*i.e.*, the $(c-1)^{\text{th}}$ first decision values are available). For a few problems such as the Atomic Broadcast problem, consensus based solutions that do not require this synchronization have been proposed [16, 8]. The specification of the *Multiple-Integrated Consensus* problem does not include this synchronization constraint and allows a chaotic construction of the sequence of decisions. However, in the protocol under study, namely Paxos-MIC, we made the choice of generating decisions in the order defined by the consensus numebr. Therefore, this design limits drastically the interest of having external Proposer-Learners that may participate simultaneously to distinct consensus instances[1].

4.2 The System model

We consider an asynchronous distributed system where processes communicate by message passing, through bidirectional, fair lossy channels. Messages can be duplicated and lost but not corrupted. Among the n processes involved in the agreement protocol, at most f processes may crash. Yet, a majority of correct processes never crash: $(f<n/2)$. To circumvent the FLP impossibility result [27], the system is extended with a leader election service. This service must ensure that eventually a single correct process is elected to be the leader until the current consensus instance ends.

[1]Indeed we show that there is no interest to launch more than 2 consensus instances in parallel.

4.3 Architecture of Paxos-MIC

4.3.1 Interaction with external proposers and learners

We consider an open distributed system where nodes, called *external Proposers/Learners* and denoted $PLext_k$, interact with a finite set of n nodes whose mission is to deliver a unique sequence of decision values by executing the Paxos-MIC agreement protocol. External proposers and learners are not involved in the convergence mechanism that is only driven by the interactions between the set of n nodes.

An external proposer/learner $PLext_k$ participates to a consensus instance by identifying it with a sequence number called the consensus number and denoted c. For each value of c such that $c \geq 1$, we assume that at least one correct node $PLext_{k_c}$ calls the function Propose(c,v_{k_c}). In doing so, this node suggests an initial value v_{k_c} that is likely to be the decided value determined at the end of consensus c. Without loss of generality, we assume that a node that provides an initial value during consensus c is implicitly interested in knowing the corresponding decision: therefore it must receive a *Decision* message that contains this decision value (denoted $< v, c >$).

The Paxos-MIC protocol manages the complete series of consensus and not just a single consensus instance. More precisely, it begins by running an initial consensus that is numbered 1. Once the first decision is obtained, the protocol continues its execution and starts immediately the next consensus instance numbered 2 and so on. Assuming that c is the number of the current consensus instance, the protocol can converge toward a new decision value $< v_x, c >$ because at least one correct participant calls the function *Propose(v_y,c)*[2]. In the Paxos-MIC protocol, an external entity does not have to know the consensus number corresponding to the last reached decision. As the protocol is responsible for archiving all decisions adopted in the past, a call to the function *Propose(v_k,c_a)* done by a participant that is unaware of being behind (*i.e.*, $c_a < c$) will receive back a stored decision value. On the contrary the initial value proposed by a participant that is either synchronized or ahead may be used during the current consensus (when $c_a = c$) or is stored to be used later (when $c_a > c$). While the consensus number managed within the Paxos-MIC protocol is incremented by 1 after each decision, an external entity can execute calls in a random order (knowing that any proposed value can potentially become a decision value). At any time, the number of consensus that a node has participated in and for which he has not yet received a decision can be greater than one: this flexibility can potentially be exploited by the distributed application.

4.3.2 Roles and communication scheme

Let us now consider the n nodes that are directly involved in the execution of Paxos-MIC. Each of them may act as a coordinator, as an acceptor, or both. The role of coordinator is played by n_c nodes while n_a nodes behave as acceptors. Among the n_c coordinators (respectively, the n_a acceptors), at most f_c coordinators (respectively, f_a acceptors) may fail by crashing. The correctness of Paxos-MIC relies on the fact that at least one coordinator is correct and a majority of acceptors never fail: $(n_c \geq f_c + 1) \wedge (n_a \geq 2f_a + 1)$. Whatever the deployment , we have $n_c + n_a \geq n \geq maximum(n_c, n_a)$. The notations C_i (with $1 \leq i \leq n_c$) and A_j (with $1 \leq j \leq n_a$) are used to refer respectively to a coordinator and to an acceptor.

[2]Of course, if a single external entity is involved in the consensus c, the values v_x and v_y are equal.

An acceptor is a passive entity that performs an update of its state whenever it receives an *Operation* message from a coordinator acting as a leader or a *Propose* message from an external proposer/learner. Regardless of whether or not its state changes, an acceptor always sends a *State* message that reflects its current state to its supported leader. Moreover when it receives an *Operation* message from a leader that is not the one it supports, it also provides its state (that includes the identity of the supported leader) to this other leader. We extend the usual remits of an acceptor with a *logging* activity: in Paxos-MIC, an acceptor also acts to guarantee the persistence of past decisions. The protocol ensures that, for each decision value, at least one correct acceptor will be aware of it. A past decision can be obtained by invoking the *RetrieveDec* function that fetches the required value from the acceptors logs. In order to integrate the optimization $\mathbf{R_O}$, we extend the acceptor's activity by allowing it to directly receive proposals from external proposers.

As in Paxos, a coordinator in Paxos-MIC is defined as an active entity that has the ability to select a value that can safely become a decision value. As in any Paxos-like protocol, an asymmetry is created between the coordinators by using a leader election service. The Paxos-MIC protocol is indulgent: it never violates its safety properties even when, at the same time, multiple coordinators consider themselves leaders. Once the role of leader is assigned to a single correct coordinator, the stability of this elected leader over a period of time that is long enough is a key element to ensure the termination of the current consensus instance. Regarding the selection of a leader, a two-stage approach that involves both the acceptors and the coordinators is adopted in Paxos-MIC.

- Each acceptor A_j periodically queries a leader election service in order to obtain the identity of the single coordinator C_{l_j} that A_j identifies as being the current leader. This information (*i.e.*, the identity of the coordinator C_{l_j} currently supported by A_j) is transmitted by A_j to the coordinator C_{l_j} and also to any coordinator who acts as a leader without the support of A_j.

- A coordinator receives information from the acceptors that support its leadership. Once a coordinator is supported by a majority of acceptors, it acts as a leader. As such it interacts with all the acceptors. Therefore if a majority of acceptors no more support its leadership, a coordinator will inevitably become aware of this and will stop being the leader.

At any time, each coordinator can determine whether it should act as a leader or not. Moreover, each acceptor identifies a coordinator that seemed to be the current leader.

Regarding the communication scheme, depicted in Figure 4.1, coordinators never communicate with each other. Similarly acceptors never interact with each other. Coordinators and acceptors use two types of messages to interact: *Operation* and *State* (denoted in Figure 4.1 by Op and St). A coordinator that is not acting as a leader remains silent. It can only receive a *Propose* message from an external proposer/learner or a *State* message from an acceptor. As a leader, it can broadcast *Operation* messages to all acceptors.

Note that this communication scheme is more or less adopted by all the Paxos like protocols. The fact that an external proposer/learner communicates directly with the acceptors is at the core of the optimization proposed in Fast Paxos [39]. The query-reply scheme implemented in Paxos-MIC is inspired from the one proposed by Lamport. A coordinator broadcast an *Operation* message and the acceptors send back *State* messages. Yet Lamport

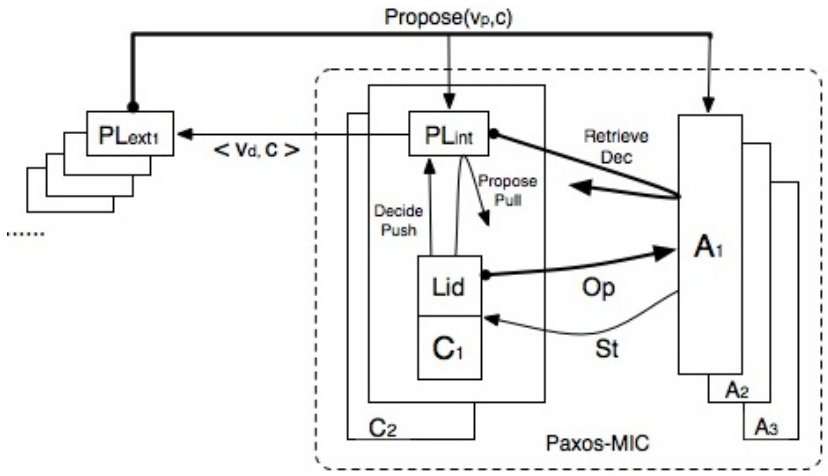


Figure 4.1: Interaction Scheme.

distinguishes a request corresponding to a *Read* operation (broadcast during a *Prepare* phase) from a request corresponding to a *Write* operation (broadcast during a *Propose* phase). In Paxos-MIC, this distinction that is managed by the leader does not appear in the messages it transmits: as explained later, each operation is interpreted as being both a *Write* operation and a *Read* operation. Thanks to this standardization of messages, all the actions performed by an acceptor and the major part of the actions performed by a coordinator, aim at maintaining their local state as up-to-date as possible by taking into account the received information when this one appears to be more recent. As a consequence, the query-reply communication pattern implemented in Paxos-MIC does not use well-formed requests. More precisely we assume neither that a coordinator initiates only one query-reply at a time, nor that it waits for appropriate replies before proceeding to the next step. To cope with message losses, the last sent message is retransmitted periodically as it reflects the current state of the sender.

To each coordinator is attached an internal proposer-learner running on the same machine (called *PLint* for short). A *PLint* is in charge of the interactions between its associated coordinator and the external proposers and learners. In Paxos-MIC, the usual specification of a coordinator is extended in order to ensure the interaction with its associated *PLint*. A coordinator never interacts with a remote *PLint*. The communication between a coordinator and its local *PLint* is performed via two functions: *ProposePull* and *DecidePush* (see Figure 4.1). Note that the call is initiated by the coordinator in both cases. The code executed by a *PLint* is described in Figure 4.2.

The *PLint* may be regarded as a temporary storage unit that knows the following:

- the current consensus number c;
- the received proposals related to this consensus instance, denoted by *Buffer*. An entry in this buffer contains a proposal for consensus c, with the value v_p, made by proposed with identity p_{id};
- the identities of the learners interested by the corresponding decision;
- the last decision value *DVal* corresponding to the previous consensus instance $c - 1$.

At any time, a *PLint* gathers proposed values related to consensus c upon the invocation of *Propose(v_p, c)* by a *PLext* (lines 1 to 4). A *PLint* may provide any decision already made. Indeed, it knows the last decision corresponding to consensus $c-1$ and it can call the *RetrieveDec* function to ask to all the acceptors a decision made during a consensus whose number is between 1 and $c-2$ (lines 5 to 14).

A coordinator calls the *ProposePull* function only when it acts as a leader. The value returned by *PLint* is either an initial value v (that may become a decision value $< v, c >$) or a special mark, $\perp$ or $\top$ (that cannot be selected to become a decision value). Indeed, *PLint* has three possible choices. It can postpone its choice until the next call to the *ProposePull* function. In that case, it uses the special mark $\perp$ to invite the calling leader to call again this function later. Otherwise, the proposer can return immediately a definitive answer that is either an initial value (if available) or the special mark $\top$: the leader will never have to call again the *ProposePull* function during this consensus instance. When an initial value v is returned, it becomes the initial value of the leader during the current consensus instance. Following the Paxos terminology, $\top$ is called an *Any* value. When *PLint* returns $\top$, it allows the use of optimization $\mathbf{R_O}$, because the context seems favorable. For example, it may return $\top$ after having returned $\perp$ during several previous calls. In that case, as all the external proposers seem to be idle, using the second optimization is more likely to be beneficial. The special mark $\top$ will be used by the leader as if it were a significant value. The acceptors will adopt temporarily this special mark $\top$ until they can replace it with an initial value v directly provided by an external proposer. During consensus instance c, once the protocol has converged to a decision $< v, c >$, a call to the *DecidePush* function made by a coordinator, provides the new decision $< v, c >$ to the local *PLint*.

When the *Propose* primitive is invoked by an external proposer (denoted *PLext*), a message containing a consensus number c and an initial value v is broadcast to all the *PLint* and all the acceptors (see Figure 4.1 describing the interaction scheme between all the entities involved in Paxos-MIC). An acceptor has to determine if it accepts or ignores each value it receives from proposers. When the optimization $\mathbf{R_O}$ is not used, the acceptors ignore all the values directly received from an external proposer: only a value that has passed through a *PLint* and an Op message can become a decision value.

4.4 The behavior of Paxos-MIC with just $\mathbf{S_O}$

Within this section, we provide a description of the protocol without considering the use of *Any* values: a call to the *ProposePull* function returns either an initial value or the special mark $\perp$ (but never the *Any* value $\top$). Under this additional assumption, the proposed protocol satisfies the specification of the MIC problem and integrates in a simple way the optimization proposed in FastPaxos [8], which we denote by $\mathbf{S_O}$. This optimization $\mathbf{S_O}$ (described previously in Section 3.8.3), tries to take advantage from long-lasting stable periods, in which the current leader seldom changes. During these periods, a current leader does not have to execute a *Prepare* phase followed by a *Propose* phase for each consensus instance. In fact, the leader executes only *Propose* phases and the execution of a *Prepare* phase is required only when a new leader is elected, in order to ensure consistency with actions performed by the previous leader.

```
% Last known decision value and last known consensus number
DVal ← ⊥;  Con ← 0;
% List of received proposals: (c, v_p, P_id)
Buffer ← ∅;

Task 1: Upon invocation of Propose(v_p, c) by PL_ext
When PL_int_i receives msg Proposal(c, v_p) from PL_ext
(1)  if(c <= Con − 2) then v_d ← RetrieveDec(c);
(2)                   send Decision< v_d, c > to PL_ext;
(3)  if(c = Con − 1) then send Decision< DVal, c > to PL_ext;
(4)  if(c >= Con) then Add(Buffer, (c, v_p, PL_ext);

Procedure DeliverDecisions< v, c >
(5)  DVal ← v;
(6)  for all(P_id ∈ Buffer s.t. Buffer.c = c) do
(7)          send Decision< v, c > to P_id;
(8)          Remove(Buffer, (c, v_p, P_id);
(9)  for all((i ≥ Con) ∧ (i < c)) do
(10)         v_d ← RetrieveDec(i);
(11)         for all(P_id ∈ Buffer s.t. Buffer.c = i) do
(12)                 send Decision< v_d, i > to P_id;
(13)                 Remove(Buffer, (i, v_p, P_id));
(14) Con ← c;

Task 2: Upon invocation of ProposePull(DVal, c) returns PVal
(15) DeliverDecisions< DVal, c − 1 >;
(16) Con ← c;
(17) PVal ← TopOrBottom;
(18) if (∃v_p ∈ Buffer s.t. Buffer.c = c) then PVal ← v_p;
(19) return PVal;

Task 3: Upon invocation of DecidePush< DVal, c >
(20) DeliverDecisions< DVal, c >;
```

Figure 4.2: Protocol executed by a PL_{int_i}.

The next section extends the protocol in order to allow the treatment of *Any* values. Recall that an *Any* values is a special value sent by a leader to all acceptors with the purpose of notifying them that for the next consensus instance, they are allowed to adopt a value supplied directly by proposers. This two-step description allows to clearly identify which parts of the protocol are impacted by the risky optimization proposed in Fast Paxos [39]. Before providing the pseudo-code of the protocol, we first discuss some underlying elements of the protocol.

As suggested by its name, the Paxos-MIC protocol follows the same main principles of any Paxos-like protocol. In the following, we describe these principles in detail and we underline how they are implemented by Paxos-MIC, by providing the pseudo-code executed

by each of the participants. In addition to the *PLint* role detailed in Section 4.3, the Paxos-MIC protocol describes the behavior of two types of entities: acceptor (denoted A_i) and coordinator (denoted C_i). An acceptor executes two tasks called Tasks A and B (Figure 4.3). A coordinator executes Task C (Figure 4.4) and, when it acts as a leader, Task D (Figure 7.12). Tasks A and C are executed upon the receipt of a message. Tasks B and D are executed periodically. All the statements contained in Task D can only be executed by a coordinator when it acts as a leader: for this reason, Task D is also called the Leader Task. In the next section, Figure 4.6 includes additional code executed by the acceptors and the coordinators to take *Any* values into account.

Messages: Coordinators and acceptors use two types of messages to interact: *Operation* and *State* messages. These messages have the same uniform structure, a process does not distinguish between them. A coordinator broadcasts its *Operation* messages to all the acceptors when it acts as a leader (Task D). An *Operation* message corresponds either to a *Read* operation (Task D, line 16), or to a *Write* operation (Task D, line 14). An acceptor sends its *State* messages to the current leader (Task A, line 5 and Task B, line 9) and also to the initiator of an operation (Task A, line 5). A *State* message informs the recipients of the current state of the acceptor and it is also intended as a (positive or negative) acknowledgment by the initiator of an operation. Note that the proposed interaction scheme is slightly different from the classical one adopted in Paxos-like protocols. In Paxos-MIC, *Read* and *Write* operations are not called "requests" because the sender does not wait for a reply. Even if an acceptor always sends a *State* message in reply to each received *Operation* message, such a query-reply communication pattern does not correspond to the well-formed requests used in Paxos-like protocols. We assume neither that a coordinator initiates only one query-reply at a time, nor that it waits for appropriate replies before proceeding to the next step. To cope with message losses, the last sent message is retransmitted as it reflects the current state of the sender. For this reason, Tasks B and D are executed periodically.

Variables: Table 4.1 provides a brief overview of the variables used in the code. It indicates, for each variable, the role played by the process that manages it (acceptor A_i or coordinator C_j), a small description of its meaning and its assigned initial value. The list of variables is divided into three parts, based on the purpose they have in the protocol. The first part of the variables is related to the use of *the leader election* mechanism, while the second part of the list comprises the variables involved in the management of *tags* and *tagged values*. Finally, the last part of the table describes the variables used for the *deciding*, *learning* and *logging* processes. Note that three variables (*SetRnd*, *SetCTag* and *SetLid*) are used by a coordinator to manage sets of acceptors identities. In the code, if X is one of the three sets, any call to the function $Reset(X, false)$ empties the set.

Leader Election: The leader election service is invoked only by the acceptors and never by a coordinator. This service ensures that a new process is selected as a leader when the current leader is suspected to have crashed and can be provided by a failure detector oracle Ω, as explained in Section 3.1. Moreover, it guarantees that eventually, all the acceptors will select the same correct coordinator to be the leader. Task B of the protocol is periodically executed by an acceptor to query this service by executing a call to the function *GetLeader* (line 8). After learning the identity of its leader, an acceptor sends a *State* message to the chosen coordinator (line 9). Indeed, the first field *St.Lid* of such a message contains the identity of the elected coordinator.

A coordinator determines if it should act as a leader by relying indirectly on the infor-

Table 4.1: Table of variables

Role	Name	Significance	Initial value
A_i	*Lid*	identity of the supported leader	1
C_j	*SetLid*	set of A_i that support $C_j$as leader	$[true, .., true]$ if $j = 1$ $[false, .., false]$ if $j \neq 1$
	RndLid	leader's round number	j
A_i	*Rnd*	highest round ever observed	1
	VTag	highest tag ever observed	(0,0,0)
	VVal	value associated to *VTag*	$\perp$
C_j	*Rnd*	highest round ever observed	1
	SetRnd	set of A_i that reached *Rnd*	[*true*,..,*true*]
	CTag	highest tag ever observed	(1,1,0)
	CVal	set of the most recent tagged values	[$\perp$,...,$\perp$]
	SetCTag	set of A_i that sent CTag	[*false*,..*false*]
A_i	*LogDVal*	array of logged values	[$\perp$,...,$\perp$]
C_j	*DVal*	last known decision value	$\perp$
	PVal	value used for a *Write* operation	$\perp$
	PreparePhase	current phase	*false*
	LVal	last recorded tagged value	$\perp$

mation supplied to the acceptors. A coordinator C_i does not store the identity of the leader selected by an acceptor A_j but just the fact that it has been chosen or not by A_j. Every time C_i receives a *State* message from A_j, $SetLid[j]$ is set to *true* only if A_j considers C_i to be the leader (Task C, line 1). C_i can act as the current leader only when it has gathered the support of a majority quorum of acceptors (Task D, line 1). After the initialization phase, the coordinator C_1 is supported by all the acceptors as the initial leader. A coordinator that has never been a leader remains quiescent until it obtains the support of enough acceptors. Each time an acceptor receives an *Operation* message from a deposed leader, a *State* message is returned (Task A). Consequently, the deposed leader will eventually discover that it is no longer supported. As a *State* message is periodically sent to the leader, a unique leader eventually obtains a stable and up-to-date information from a majority of acceptors, once all the old messages have been lost or consumed.

In order to ensure progress, the leader election service must guarantee that, for all the acceptors, any call to the *GetLeader* function eventually designates the same correct coordinator. As a *State* message is periodically sent to the leader, a unique leader eventually obtains this stable and up-to-date information from a majority of acceptors, once all the old messages have been either lost or consumed The Paxos-MIC protocol is indulgent as it never violates its safety properties even if, at the same time, multiple coordinators consider themselves leaders.

Consensus Instances and Round Periods: A coordinator proceeds in a sequence of consensus instances and also in a sequence of round periods. Each consensus instance (respectively, each round period) is identified by a consensus number, variable *CTag.Con* (respectively, a round number-variable *Rnd*). These counters are independently updated. The

```
Task A: When A_i receives Op(Rnd, Tag, Val, DVal) from C_j
        % Maintaining the most recent information ever observed
(1)     if ((Op.Tag.Rnd ≥ Rnd) ∧ (VTag ≺ Op.Tag) ∧ (Op.Val ≠ ⊥))
(2)         then VTag ← Op.Tag;  VVal ← Op.Val;
(3)     if (Op.Rnd > Rnd) then Rnd ← Op.Rnd;
        % Logging decision values
(4)     LogDVal[Op.Tag.Con - 1] ← Op.DVal;
(5)     send St(Lid, Rnd, VTag, VVal, LogDVal[VTag.Con - 1]) to C_j and C_Lid;

Task B: Periodically
        % Query the Leader Election Service
(6)     Lid ← GetLeader();
(7)     send St(Lid, Rnd, VTag, VVal, LogDVal[VTag.Con - 1]) to C_Lid;
```

Figure 4.3: Protocol executed by an acceptor A_i.

division into consensus instances follows from the specification of the problem itself. The division into round periods results from the use of a leader election mechanism: a leader starts a new round period when it discovers that another coordinator has concurrently acted as a leader by executing a round period with a higher round number. Consensus instances and round periods are not linked: a consensus instance may span over several round periods and, conversely, during a single round period, several consensus instances may be solved (when optimization $\mathbf{S_O}$ is used). The current consensus number (*CTag.Con*) increases only when a new decision is made during Task C (line 17). Indeed, the participation of a coordinator C_i to a new consensus instance numbered c, can only start when the outcome of the previous instance, $c-1$, is obtained. The current round number (variable *Rnd*) is increased either during Task C (line 9) when a coordinator observes a higher round or during Task D (line 3), when a leader starts a new round period. More precisely, the round number of a coordinator has to satisfy the three following rules. At any time, the round number of a coordinator (*Rnd*) must be greater or equal to the highest round ever observed (variable *CTag.Rnd*). Therefore, when it receives a message, a coordinator may adopt a round number already reached by another coordinator (line 9). When C_i is elected as a new leader, it has to adopt the round number under which it will execute attempts to converge towards decision values (Task D, line 4). This round number has to be strictly higher than the highest round ever observed by C_i. Moreover, another leader should not be able to use the same number. In the code, the variable *RndLid* is used by C_i, to identify this particular round number. By construction (Task C, line 10), the value of *RndLid* is obtained by adding the value i to a multiple of n. Thus, a round period numbered r is associated to a unique leader whose identity is equal to $r \bmod n$. While it acts as a leader, the two variables *Rnd* and *RndLid* managed by a coordinator are equal (Task D, lines 2 and 4). When the coordinator is not acting as a leader, the variable *RndLid* may be strictly greater than the variable *Rnd*.

Prepare and Propose Phases: As in Paxos [38], a round period consists of an initial *Prepare* phase followed by a *Propose* phase. This distinction is only relevant when a coordinator acts as a leader (*i.e.* during an execution of Task D). Otherwise (*i.e.* during an execution of Task C), a coordinator takes its current round period number into account, but ignores the subdivision into two phases. A boolean variable *PreparePhase* indicates the current phase.

This variable is set to true at the beginning of any round period and remains true till the *Propose* phase can start. After the initialization, the coordinator C_1 can immediately execute the *Propose* phase of the round period number 1. Apart from this particular optimization, any leader executing the *Propose* phase of a round has previously executed the corresponding *Prepare* phase. Similar to any Paxos-like protocol, during the *Prepare* phase (respectively, *Propose* phase), a leader can generate *Read* (respectively, *Write*) operations, while it executes Task D.

The names given to the phases and the operations are similar to those used in the Paxos terminology introduced by Lamport [38]. The purpose of a *Prepare* phase is to ensure that all the future *Write* operations of a leader will be consistent with those performed by previous leaders. During this phase, a leader communicates its new round number to the acceptors (*Read* operations). If it gathers enough feedbacks from them, it can switch to the *Propose* phase (Task D, line 8). When this phase transition occurs, the leader uses the information collected during the *Prepare* phase to determine the origin of the value it will use during the first *Write* operation of its *Propose* phase (variable *PVal*). Two cases are envisioned. If at least one acceptor has informed the leader of a value previously proposed by another coordinator acting also as leader during the same consensus instance, the leader must select one of the most recent values among such values (Task D, lines 7-8). In the basic version of the protocol, as we assume that no *Any* value is proposed, all these tagged values are necessarily equal and different from $\perp$ and $\top$. Otherwise, at the end of the *Prepare* phase, the value of *PVal* remains equal to $\perp$. As any value can be chosen, the proposed value can be provided later by the *PLint* in reply to a call to the *ProposePull* function (Task D, line 12).

```
Task C: When C_i receives St(Lid, Rnd, Tag, Val, DVal) from A_j
(1)   SetLid[j] ← (St.Lid = i);
      % Maintaining the most recent information ever observed
(2)   if (CTag ⪯ St.Tag) then
(3)      if (CTag ≺ St.Tag) then
(4)         if (St.Tag.Con < CTag.Con) then DVal ← St.DVal;
(5)         CTag ← St.Tag;  Reset(SetCTag, false);
(6)      CVal[j] ← St.Val;  SetCTag[j] ← true;  LVal ← St.Val;
(7)   if (St.Rnd ≥ Rnd) then
(8)      if (St.Rnd > Rnd) then
(9)         Reset(SetRnd, false);  Rnd ← St.Rnd;
(10)        while (RndLid < Rnd) do RndLid ← RndLid + n;
(11)     SetRnd[j] ← true;
      % Deciding for the current consensus instance
(12)  if (CTag.Any = 0) then
(13)     if ((Quorum_Maj(SetCTag) ∧ (LVal ≠ ⊤)) then
(14)        DVal ← LVal;  PVal ← ⊥;  LVal ← ⊥;
(15)        DecidePush(< DVal , CTag.Con >);
(16)        Reset(SetCTag, false);
(17)        CTag.Con ← CTag.Con +1;  CTag.Any ← 0;
(18)  else execute(code CAny);
```

Figure 4.4: Protocol executed by a coordinator C_i.

During the following *Propose* phase, each *Write* operation executed by the leader aims at suggesting a safe value to the acceptors. If enough acceptors follow this suggestion, this value becomes the decision value. Once the decision is learned (during the execution of Task C), the protocol goes on with the next consensus instance. While the leader remains stable, it will continue to execute the same round period and, more precisely, the same *Propose* phase. The fact that several consensus instances can be executed during the same round period is the result of the optimization $\mathbf{S_O}$. Let us notice that, once it enters the *Propose* phase, a leader broadcasts at most one *Write* message during each execution of Task D. Indeed, if it can propose neither a significant initial value nor an *Any* value $\top$, a leader sends no message. Otherwise, it periodically sends its last *Write* message until it makes a decision during Task C. Each time it decides, the proposed value is reset to $\perp$, the consensus number is increased and all the tagged values are discarded.

Tags and Tagged Values: A *tag* is defined as a triplet of integers denoted (r, c, a): the first integer r is a **R**ound number, the second integer c is a **C**onsensus instance number and the last integer a is only useful when **A***ny* values are used. At this stage of the explanation, we just need to know that this last integer is effectively a boolean variable, set to 0 or 1. When no *Any* values are used, a is always equal to 0. Otherwise, it can be set to 0 or 1. A *tagged value* v is defined as the result of a deliberate decision to associate a value v with a tag (r, c, a) and we represent this through the notation $(v, (r, c, a))$. In Paxos-MIC, we assume that a tagged value can be created (or declared) only at some well-defined stages of the computation, namely during the initialization phase and during a *Write* operation. Indeed, only the leader is allowed to declare new tagged values during the computation. Once it has been declared, a tagged value is propagated within the set of processes. A tagged value is contained in the *Operation* messages broadcast by a coordinator to all the acceptors and in the *State* messages sent by an acceptor to some coordinators. When a message is received, a copy of the transmitted tagged value can be stored temporarily in a set of four related variables. With regard to a given consensus instance, an acceptor can store a single tagged value while a coordinator can store up to n tagged values that share the same tag. More precisely, an acceptor stores a tagged value in its variables ($VVal$,($VTag.Rnd$,$VTag.Con$,$VTag.Any$)). This set of variables is initialized to $(\perp, (0, 0, 0))$. A coordinator can store a tagged value received from an acceptor A_j, in its set of variables ($CVal[j]$,($CTag.Rnd$,$CTag.Con$,$CTag.Any$)). This set of variables stores a tagged value if and only if the variable $SetCTag[j]$ is equal to true. When no *Any* values are used, all the values logged in $CVal$ by a coordinator are equal. Consequently, this common value is also contained in the variable $LVal$, which is used to keep the last recorded tagged value.

Lexicographical order: A lexicographical order is defined over the set of tagged values and denoted $\preceq$. Let $(v, (r, c, a))$ and $(v', (r', c', a'))$ be two tagged values. Then, $(v, (r, c, a)) \preceq (v', (r', c', a'))$ if and only if $(r < r') \vee ((r = r') \wedge (c < c')) \vee ((r = r') \wedge (c = c') \wedge (a \leq a'))$. When $(v, (r, c, a)) \preceq (v', (r', c', a'))$ and $(r, c, a) \neq (r', c', a')$, the tagged value $(v', (r', c', a'))$ is said to be more recent than $(v, (r, c, a))$, denoted by $(v, (r, c, a)) \prec (v', (r', c', a'))$. When a process receives a message which contains a tagged value, the tag of the received value and the tag of the value(s) already stored, are compared to determine the most recent one. Indeed, the coordinators and the acceptors only keep the most up-to-date tagged values, by taking into account only the messages that provide a more recent information. Task A executed by an acceptor, manages the updating mechanism. When an acceptor receives an *Operation* message from a leader (Task A), it updates its tagged value

(variables *VTag* and *VVal*) at lines 1-2 and its round number (variable *Rnd*) at line 3. The other fields of a *State* message are similar to those of an *Operation* message. In addition to the leader identity, a *State* message includes a round number r, a tagged value $(v_p, (r_p, c, a))$ and a value v_d. The value r contained in the variable *Rnd* matches the highest round number ever observed by the acceptor. The tagged value $(v_p, (r_p, c, a))$ contained in the variables (*VVal*, (*VTag.Rnd* , *VTag.Con* , *VTag.Any*)) is the most recent one that has been accepted. The value v_d is the last known decision value obtained during the previous consensus instance numbered $c - 1$. By construction, this value is stored in the variable *LogDVal*[*VTag.Con - 1*]. A coordinator follows a similar behavior: during Task C, a coordinator updates its tagged values at lines 2-5 and its round number at lines 7-9. At any time, $SetCTag[j]$ is equal to true if and only if the coordinator has received from A_j the value contained in $CVal[j]$ associated with the tag *CTag*. Recall that, the variable *LVal* is used to keep the value of the last recorded tagged value. When a coordinator observes a higher tag, it resets the variable *SetCTag* before recording the value associated to the new tag. When a coordinator updates the value of the highest round number it has observed (variable *Rnd*), it also resets the list of acceptors that have reached this level (variable *SetRnd*). Note that the behavior of the acceptor is more restrictive because the test performed in Task A, at line 1 may lead to ignore a tagged value even if it is more recent than the logged one.

When they are not lost, messages may arrive in a random order at their recipients. As coordinators and acceptors manage monotonically increasing variables (*Rnd*, *RndLid*) and ordered tagged values (*CTag*, *VTag*), a receiver can easily ascertain which fields of the message provide a more recent information. Note also that while the first field of a tag (the round number) is a monotonically increasing variable, the second field (the consensus number) may vary non-monotonically due to the use of the $\preceq$ relation. The computation of *RndLid* ensures that the leader's tagged value will be the most recent one in the system. Thus, its proposed value has a chance of being adopted by at least a majority of acceptors.

Operations: Only a leader is allowed to initiate an *Operation*. An *Operation* message includes four fields: a round number r, a tag (r_p, c, a), a value v_p, and a value v_d. The value v_d contained in the last field is the last known decision value of the previous consensus instance, $c - 1$ (*i.e.* $< v_d, c - 1 >$ is a decision). The value r, stored in the variable *RndLid*, is the current round number of the leader.

During a *Prepare* phase, the value r is necessarily strictly higher than r_p. A leader may execute several *Read* operations (Task D, line 16) during the same phase but all referring to the same round period. If an acceptor receives a *Read* operation and adopts r, it can no more consider tagged values with a lower round number, even if the received value is more recent than its current one. In a *Read* operation, the leader also relays a tagged value previously observed. Indeed, the tag (r_p, c, a) is contained in *CTag*, the highest tag ever observed. The tagged values associated with this tag are logged in *CVal* and in particular the value v_p is stored in *LVal*. As the tagged value $(v_p, (r_p, c, a))$ was previously observed by an acceptor, it had been proposed in a past *Write* operation by the leader of round r_p. The leader of round r observed it and forwards it during its *Read* operation. Task A treats an *Operation* message without checking its type (Read or Write). Roughly speaking, each received message is considered as a *Write* (lines 1 - 2) and then as a *Read* operation (line 3). Therefore, a simple test can distinguish a *Read* from a *Write* operation. In the former case, $r > r_p$, while in the latter, $r = r_p$. Note that in the beginning, due to the initialization, a leader may provide a tagged value equal to $(\perp, (1, 1, 0))$ which will not be considered by the

acceptors, during Task A.

The tagged value $(v_p, (r_p, c, a))$ contained in a *Write* message is proposed by the leader itself to become the next decision value. The value v_p is contained in the variable *PVal* and the associated tag is defined by the triplet (*RndLid*,*CTag*.*Con*,0). The *Prepare* phase executed at the beginning of a round period ensures the correctness of the first *Write* operation. At the end of the *Prepare* phase, *PVal* might have been updated to be consistent with any previous attempts to converge to a decision value, made by another leader (Task D, line 8). Once the leader has decided during round period r, the following consensus instances cannot be in conflict with a previous attempt made by another leader. C_i may participate within the same round period to several consensus instances without executing again *Prepare* phases (optimization S_O).

```
Task D: Periodically
(1)   if (Quorum_Maj(SetLid)) then  % C_i can act as a leader
(2)     if (Rnd < RndLid) then  % A new round period starts
          % Beginning of a Prepare phase
(3)       PreparePhase ← true;  Reset(SetRnd, false);
(4)       Rnd ← RndLid;
(5)     if (CTag.Any = 0) then
(6)       if ((Quorum_Maj(SetRnd) ∧ PreparePhase) then
            % C_i has gathered enough feedbacks from acceptors
            % It selects one of the most recent values
(7)         if (∃ k s.t. SetCTag[k]) then
(8)           PVal ← CVal[k];  PreparePhase←false;
            % C_i can now execute the Propose phase
(9)     else execute(code DAny);
(10)    if (PreparePhase = false) then
          % A Write operation must contain a non-⊥ value
(11)      if (PVal = ⊥) then
(12)        PVal ← ProposePull(DVal, CTag.Con);
(13)      if (PVal ≠ ⊥) then  % C_i executes a Write operation
(14)        send Op(RndLid, (RndLid, CTag.Con, 0), PVal, DVal) to every A_k;
(15)    else  % C_i executes a Read operation
(16)      send Op(RndLid, CTag, LVal, DVal) to every A_k;
```

Figure 4.5: Protocol executed by a leader C_i.

Decisions: Let us consider a coordinator C_i that receives a *State* message such that the tag contained in the field *St*.*Tag* is equal to the highest tag ever observed by C_i and stored in *CTag*. C_i can decide if it observes that a majority quorum of acceptors have adopted tagged values during the same round and consensus instance (Task C, lines 13 - 17). The test is done when a *State* message is received. As no *Any* value is used, all these tagged values are necessarily equal and different from $\perp$ and $\top$.

Logs: By construction, the field *DVal* of a *State* or an *Operation* message, related to consensus instance c, contains the last known decision value, corresponding to the previous consensus instance $(c-1)$. Old decision values are logged by the acceptors. Each acceptor A_i maintains an array of logged values, *LogDVal*. An entry k of this array is initialized

to $\perp$ and used to store the decision for consensus number k. A_i may acquire this value which is contained in the field *DVal* of any *Operation* message related to consensus number $k+1$. To decide a value v during consensus $k+1$, a coordinator must observe that v has been adopted by a majority of acceptors. Consequently, the previous decision contained also in these *Operation* messages, has necessarily been observed and logged by a majority of acceptors. At least one of them is correct and can provide it, if necessary. The *RetrieveDec* function is used by an internal learner *PLint* to obtain an old decision value. In our solution, each time an acceptor sends a *State* message during consensus number c, it includes in the last field of the message, the logged value related to consensus $c-1$. Due to this choice, *Operation* and *State* messages have similar structures. Moreover, it speeds up the retrieving of the last decision value. The logs can be used to ensure the termination property which states that, during each consensus instance, at least one correct process eventually decides a value. Of course, in an asynchronous system, these logs might store an unbounded number of values. If weaker termination properties are considered, it is possible to implement amnesic logs that store only a limited number of decision values [20].

A different mechanism for logging decisions is used in [8]. A distributed structure, called a *round based register*, is used to log decision values. Each time a new consensus instance is started, a new register is created. Any correct process must be able to retrieve the decision value for any completed consensus instance. For achieving this purpose, the register instances must remain active even after the corresponding consensus instance has finished. The logged information is available and can be retrieved at any time. Such a logging mechanism requires that (possibly) a high number of register instances are kept active.

Retrieving Decision Values:

The retrieving mechanism is employed by a *PLint* whenever it needs to provide a decision value to a *PLext* that is interested in learning a previous decision. The *RetrieveDec* function takes as parameter, a consensus number c and returns the decision value corresponding to consensus number c. This function can be invoked by external learners or by a *PLint* executing lines 1 or 10. The retrieving mechanism fetches decision values from the acceptors logs. A call to the *RetrieveDec* function with c as parameter, will periodically broadcast a request to obtain the decision value corresponding to consensus number c. This request message is sent to at least a majority of acceptors. Each acceptor A_j store the decision value for consensus number c, in the entry c of its logs (variable $LogDVal[c]$), only if A_j has received this value in an *Operation* message sent by a leader. If the message has not reached A_j, $LogDVal[c]$ stores the $\perp$ value. Each time A_j receives a request to access the c entry of its logs, it will provide the corresponding value only if this value is different from $\perp$. Once a *PLint* receives a message containing a value different from $\perp$, it has successfully retrieved decision number c.

4.5 Paxos-MIC with both S_O and R_O

We present the modifications that have to be done in order to integrate the use of *Any* values. During a call to the *ProposePull* function, a *PLint* can now take the decision to use the optimization O_2, by returning $\top$ (see Section 4.3). Proposers make a commitment to provide later all the acceptors with at least one significant value. Such a value is sent directly by a proposer to acceptors and it is called a *direct* value. The name *Any* value denotes either the

special mark $\top$ or a direct value received by an acceptor from an external proposer. Detecting that v is an *Any* value is obvious when v is equal to $\top$ but impossible when v is a direct value. To solve this, the third field of a tag, called *Any*, is set to 1 when the value adopted by an acceptor is a direct value. This boolean field only indicates the origin of the value (provided by an external proposer or by a coordinator). If the leader is currently executing the *Propose* phase of a round period r, when it obtains a $\top$ value, it can instantly broadcast *Operation* messages with the tagged value $(\top, (r, c, 0))$. An acceptor may adopt the $\top$ value as if it were a real initial value.

As the $\top$ value is managed like an initial value, we have to ensure that no coordinator decides $< \top, c >$, in Task C at line 15. The test performed at line 13 was introduced for this purpose. To allow an acceptor to receive a direct value from a proposer, we define an extension of the acceptor's protocol. In Figure 4.6, the additional code is denoted by Task A-Any and it is triggered by the reception of a Propose message from an external proposer. This message contains a consensus number c and a direct value that should be an initial value (different from $\perp$ and $\top$). If the current tagged value $(v', (r', c', a'))$ of the acceptor is such that v' is equal to $\top$, $c' = c$, and r' is still the highest round number ever observed by the acceptor, then the direct value replaces the $\top$ value and the *Any* field of the tag (*VTag.Any*) is set to 1. This verifies that the last operation modifying the state of an acceptor, was the *Write* of the $\top$ value.

Different proposers may provide different direct values. These proposal messages may be received in different orders by different acceptors. In such cases, *a collision* may occur. Unfortunately, all these values will share the same tag $(r, c, 1)$. As indicated by Lamport [39], majority quorums have to be replaced by larger sets called *Fast* or *Any* quorums. In [39], Lamport defines quorums as sets of acceptors. Each round has a set of quorums associated with it: classic rounds use classic quorums while fast ones rely on fast sets. *The Quorum Requirements* state that (1) any two quorums must intersect and (2) any quorum and any two fast quorums from the same round must also have a nonempty intersection. Based on these requirements [30], Lamport has defined in [39], the minimum number of acceptors contained in a quorum. Let N be the total number of acceptors in the system. The cardinality of any classic quorum, Q_c, and of any fast quorum, Q_a, can be computed as follows: $|Q_c| \geq \lfloor N/2 \rfloor + 1$, $|Q_a| \geq \lceil 3N/4 \rceil$.

The same results are also obtained in [60]. As an *Any* quorum is larger, the maximal number of tolerated failures f has to be lower (or timeout mechanisms have to be used to prevent a deadlock). These larger quorums are more difficult to obtain as they require to collect more replies from the acceptors. In Paxos-MIC, an *Any* quorum is mandatory if and only if the highest tag ever observed by the coordinator is associated to a direct value. The quorum used in Task C, line 13 (during the learning activity) should be an *Any* quorum when the current tag is associated to direct values (code CAny is executed). Otherwise a majority quorum can be used. Note that the quorums used in Task D, line 1 and in line 6 (during the *Prepare* phase) are still majority quorums.

When an *Any* quorum is used, the selection of a value among the set of received tagged values should return one of the most frequent values. No collision occurs when only one external proposer is active or when all external proposers provide the same *direct* value: all the values logged in *CVal* are equal. This unique value can be decided (code CAny, lines 6 - 10). It could also be the case that one particular direct value is frequent enough so that it can be safely chosen. Indeed, the predicate *CollisionSafe* allows to implement different strategies.

```
Task A-Any: When A_i receives Propose(Val, Con) from PLext_k
(1)   if ((P.Val ≠ ⊥) ∧ (P.Val ≠ ⊤) ∧
(2)      (VVal = ⊤) ∧ (VTag.Con = P.Con) ∧ (Rnd = VTag.Rnd))
(3)      then VTag.Any ← 1;  VVal ← P.Val;
(4)           send St(Lid, Rnd, VTag, VVal, LogDVal[VTag.Con-1]) to C_Lid;
code CAny:
(5)   if (CollisionSafe) then
(6)      if (Quorum_Any(SetCTag)) then
(7)         DVal ← the most frequent value CVal[k]
                    such that SetCTag[k];
(8)         PVal ← ⊥;
(9)         DecidePush(< DVal , CTag.Con >); CTag.Any ← 0;
(10)        Reset(SetCTag, false);  CTag.Con ← CTag.Con +1;
(11)     else if (Rnd = RndLid) then RndLid ← RndLid + n;

code DAny:
(12)  if (Quorum_Maj(SetRnd) ∧ PreparePhase) then
(13)     PVal ← the most frequent value CVal[k]
                 such that SetCTag[k];
(14)     PreparePhase←false;
```

Figure 4.6: Extension to the protocol to cope with *Any* values.

In its simpler version, this function may return true when all the gathered values are equal and false otherwise. In the code CAny, when *CollisionSafe* is false due to the existence of too many collisions, possible deadlocks are avoided by forcing the execution of a new *Prepare* phase, if the coordinator is the current leader (code CAny, line 11). The leader remains the same and it selects again a new *PVal* value among the multiple direct values.

Finally, note that during the *Prepare* phase executed by a new leader, the selected value can be the value $\top$. When this occurs, the value of the variable $CTag.Any$ is equal to 0 and all the tagged values stored in $CVal$ are equal to $\top$. As the $\top$ value is selected at line 8, in Task D, a new leader will execute a *Write* operation with an *Any* value $\top$ like a previous leader, but with a higher round number. Proposers are expected to provide again their direct values until the end of this consensus instance.

In [39], Lamport suggests to optimize the classical mechanism to recover from collisions. However, this optimization is only possible under stronger assumptions. If i is a fast round and C_i is coordinator of rounds i and $i+1$, the information last sent during round i can be used during round $i+1$. Based on this observation, C_i can skip the *Prepare* phase for round $i+1$ as it knows that no one else has acted as a leader between rounds i and $i+1$. Therefore, round $i+1$ can begin directly with the *Propose* phase. Two *collision recovery* mechanisms are described in [39], namely *coordinated* and *uncoordinated* recovery.

4.6 Paxos-MIC: positioning with respect to the Paxos protocol

Although it follows the Paxos approach, the Paxos-MIC protocol includes numerous different features compared to the original protocol. As stated before, the Paxos-MIC protocol allows to solve an infinite sequence of consensus instances. In [38], the description of Paxos focuses mainly on a single consensus instance.

In this section, we highlight the aspects that differentiate Paxos-MIC from the Paxos approach.

1. The main feature of the proposed protocol is represented by its adaptability. The decision to activate optimization $\mathbf{R_O}$ is taken by the current leader dynamically and locally, at runtime and depending on the current context. This triggering of the optimization does not require any synchronization between the leader and other actors. In the original Paxos [39], the use of the optimization is decided before the execution of the protocol itself begins. More precisely, there exists an *a-priori* static convention between the process. This convention regards the round numbers during which the current leader is allowed to initiate a fast round.

2. The Paxos-MIC protocol ensures the persistency of all the past decision values, without sending additional messages. All decision values are logged by all acceptors in their local memories and can be retrieved at any time by learners. The logging mechanism ensures that at least one correct acceptor is able to provide any past decision value. Ensuring the persistency of the decisions allows to tolerate asynchrony between the external proposer/learners and the core members. The simple solution adopted in Paxos-MIC just requires to carry two values in each message rather than a single one.

3. The Paxos protocol distinguishes a request corresponding to a *Read* operation (broadcast during a *Prepare* phase) from a *Write* request broadcast during a *Propose* phase). In Paxos-MIC, each operation sent by the leader is interpreted as being both a *Read* and a *Write* operation. This standardization of messages leads to a uniform behavior of both acceptors and coordinators. Indeed, most actions performed by an acceptor or a coordinator aim at updating their local state when the received message contains more recent information.

4. The proposed protocol extends the concept of round in order to cope with a sequence of consensus instances. Paxos-MIC introduces the notion of *tag* that contains, in addition to the round number, the current consensus instance. This extra counter provides the index of the current consensus instance in the sequence of decisions. All the exchanged values are tagged. The lexicographical order is used to identified the more recent tagged values.

5. As a consequence of the two previous features, the communication pattern implemented in Paxos-MIC does not use well-formed requests. More precisely, the request-reply pattern is relaxed in the following way: we assume neither that a coordinator initiates only one query-reply at a time, nor that it waits for the appropriate answers before proceeding to the next step. To cope with message losses, the last sent message is retransmitted periodically as it reflects the current state of the sender.

6. The leader election service differs from the original Paxos, by indirectly relying on the information supplied to the acceptors. The leader election service is invoked only by the acceptors and never by a coordinator. In this way, a coordinator is only aware if it can behave as the current leader or not. The identity of the leader is not known by the coordinators. By moving the leader election service at the acceptors level, the communication scheme between coordinators and acceptors is impacted. Whenever its current state changes, an acceptor sends a message only to its current leader and to the initiator of an operation (usually, they are the same process). An acceptor does not need to broadcast its state to all coordinators, to ensure that the leader will receive its message, as it is done in the original Paxos. Thus, the number of messages exchanged between entities is reduced. Regarding optimization $\mathbf{R_O}$, once an acceptor adopts a value provided directly by a *PLext*, it sends its updated state to its current leader. In Fast Paxos [39], at the end of a fast round, acceptors broadcast the adopted values to all learners.

Part III

Implementation and Evaluation

Chapter 5

Implementation

Contents

THE algorithm and data structures of Paxos-MIC presented in chapter 4, are implemented in Java on top of EVA [12], an event-based distributed framework. This chapter discusses the implementation of the protocol in two steps: first, we provide an overview of the event-based architecture, EVA; in a second step, we show how abstractions defined in EVA are used as building blocks for the Paxos-MIC protocol.

5.1 EVA: An EVent based Architecture

EVA [12] was developed as part of the INRIA Gforge project PROMETEUS. EVA represents an event-based distributed framework that provides an environment for designing high-level communication protocols, (*i.e.* protocols providing complex services such as reliable atomic broadcast). The EVA framework implements a publish-subscriber communication environment that allows to structure the entities composing high-level protocols. The architecture of EVA relies upon the *event channel* abstraction for the interaction between entities that share the same address space. This abstraction facilitates a flexible and simple design of communication protocols and allows to circumvent the drawbacks of the layered-based approach traditionally used to develop these protocols.

5.1.1 The architecture of EVA

In [12], the authors propose an object-oriented architecture based on the event channel abstraction and denote it by EVA. A distributed application constructed on top of EVA is defined as a set of cooperating objects, called *components*, that communicate mainly by exchanging special objects, called *events*, via an *event channel* and also by invoking special operations, called *services*. The interaction between components is either asynchronous (by the means of events) or synchronous (by the means of services).

An entity in EVA may act as both a producer and a consumer of events and may be subordinated to any number of components. Any entity is first required to register or subordinate itself to a corresponding component before it can consume and produce events. The component is in charge of managing an event channel that has the role of routing the events produced by a subordinated producer to all interested subordinated consumers. The event channel decouples suppliers from consumers yielding the sought flexibility. Components provide an interface that allows the subordinated entities to register/access the services they wish to offer/use to/from other entities that share the same component. A component can register the services defined inside it, to a super-component to which it is subordinated. This allows the access to the component's services for external entities that share the same super-component. Components have two main roles: *i)* they control the interactions between subordinated entities and *ii)* they provide a way of structuring applications into related entities, while hiding them from unrelated entities and yielding application design more modular. Figure 5.1 depicts the architecture of a super-component (*Component E*) composed of two sub-components (*Component C* and *Component D*), an entity (*Entity X*) and an event channel.

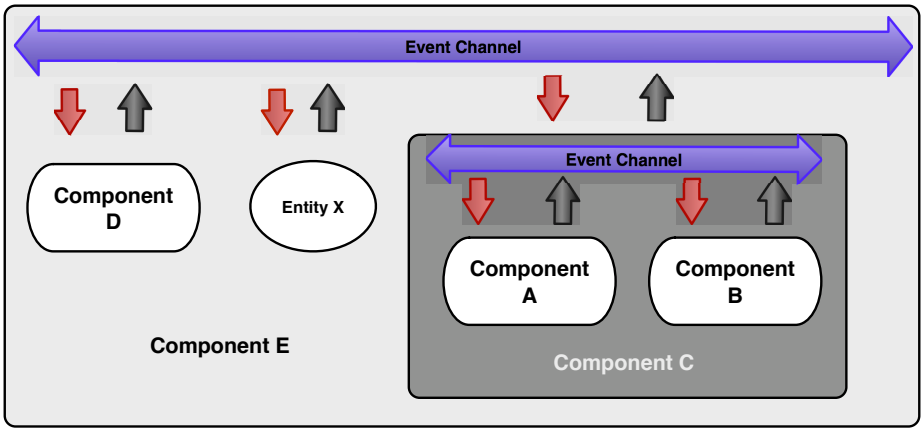


Figure 5.1: Event Channel in EVA.

In summary, the event-based architecture relies on the following four main concepts: *i)* events; *ii)* entities; *iii)* components; and *iv)* services. An event is a "container" object used to convey data from supplier to consumer entities, while a service allow synchronous communication. An entity is a "worker" object that interacts with other entities to implement part of the functionality of the application. A component is both a "structuring" object that assembles related entities together, as well as a "manager" object that coordinates the interactions of its subordinated entities. A component is itself an entity that can be subordinated to other super-components.

In the following, we detail each of the main concepts.

Events

The event concept is well suited to model messages that contain data fields. Typed events convey data that has to be exchanged between entities. In EVA, an event instantiates a class that implements the interface *Event*. This interface defines a method *getType()* to obtain the type of the event at runtime. The *EventImpl* class provided by the framework, offers a standard implementation for events. Usually, a particular application requires the definition of new types of events. This is achieved by defining classes that inherit from the standard *EventImpl* class and that add extra data fields and methods required by the application. The framework also defines interfaces that describe the events that will be consumed (*ConsumableEventDescriptor*), more precisely the way these events will be consumed, or produced (*SupplyableEventDescriptor*).

Entities

Entities provide the core functionality of the framework. An entity is an instance of a class that extends the standard class *Entity*. Furthermore, consumer/supplier entities must implement the *Consumer* respectively *Supplier* interfaces. The *Consumer* interface defines the *consumeEvent* method that is invoked by the event channel of a component when an event is delivered to the target consumer entity. The *consumptionNotification* operation defined by the *Supplier* interface, notifies a producer that new consumers are interested by its produced events.

Entities are also described as being *passive* or *active*. An active entity has an independent thread associated with it. For example, most of the consumer entities are active entities. Due to the fact that event consumption is asynchronous, each consumer buffers events for future processing. The invocation of the *consumeEvent* operation on entities of these classes simply places the given event into their buffer. The associated thread continuously verifies if new events have appeared in the buffer. It is also possible to associate with an entity, a timeout that triggers the execution of a periodic task. The timeout defines the time interval between two consecutive executions of the operation. Such entities are called *periodical* entities.

Components

A component is defined by a set of entities sharing a same event channel through which they communicate by exchanging events. A component instantiates the *ComponentEntity* class. In order to be able to use the event channel service of a component, an entity must subordinate itself to the corresponding component. This is achieved by invoking the *addEntity()* operation of the component interface, giving as a parameter a reference to itself.

A component can also behave as an entity that consumes and produces events. When a component registers itself with one of its super-components for consuming/producing a particular type of event, it registers all its internal entities that registered themselves for the consumption/production of that type of event.

Services

When entities execute within the same address space, synchronous interactions are modeled by classical procedure-call mechanisms. However, if an entity invokes directly an operation implemented by another entity, a static coupling is created between them. In order to overcome this constraint, EVA allows to decouple entities that need to interact synchronously, thus increasing the design flexibility for an application. In EVA, components provide an in-

terface that allows their subordinated entities to register the operations they want to make available to other entities sharing the same component. This is achieved by a call to the *registerService* method. Components also provide the appropriate interface to allow access to the registered operations, by a call to the method *invokeRequest*. An entity that uses an operation does not need to know the provider of the operation. Such operations are called *services*.

Services model synchronous (blocking) one-way interactions between entities. This can be exploited by applications based on a client-server communication pattern.

Inter-process communication

As mentioned before, in the EVA architecture, communication is possible between entities that share the same address space and it relies upon the event channel mechanism.

Special entities (listener/notifier) dedicated to network communication are provided by the EVA framework to allow events transmission between remote entities (located at different nodes). The framework defines special types of consumers, named *network notifiers*, able to transmit special type of events, called *remote events* from one process to another remote process. Also, special types of suppliers (*network listeners*), are defined at the corresponding receiver process. These special suppliers are responsible for receiving remote events and producing them locally. A pair of linked network notifier and network listener can be regarded as a single distributed entity that consumes an event produced by a component located on a site, and produces it at a component located on a remote site. This special type of pairs establishes the connections between distributed components. Figure 5.2 illustrates how communication is achieved between entities located remotely.

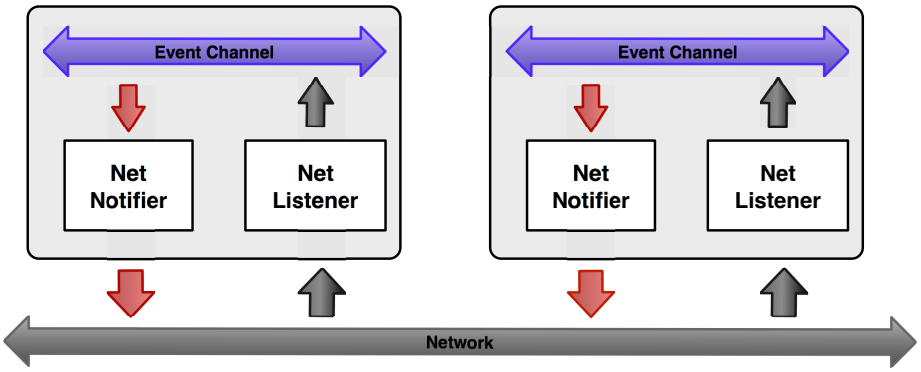


Figure 5.2: communication between remote entities in EVA.

EVA defines three basic types of notifiers and listeners:

1. *DatagramUnicastNotifier* and *DatagramUnicastListener* provide a point-to-point unreliable communication service (relying on UDP).
2. *DatagramMulticastNotifier* and *DatagramMulticastListener* implement an unreliable one-to-many communication service (UDP).
3. *StreamNotifier* and *StreamListener* provide a reliable, FIFO, point-to-point communication service (TCP).

Applications can use the provided network entities or define classes that extend them, in order to meet their particular needs. The events that are transferred across the network,

must inherit from the *RemoteEvent* class, defined by the framework. As opposed to a local event, a remote event must incorporate the definitions of the operations that marshall and unmarshall an event. These operations allow the reconstruction of an event at the remote destination.

5.2 Paxos-MIC implementation

The functionalities that EVA provides have proven to be particularly useful for building the Paxos-MIC protocol. The event-based communication model used by EVA components helps to fill in the gap between the specification of the protocol and its implementation.

5.2.1 Building the components

We first identify the entities that need to be constructed. The implementation of Paxos-MIC defines four main classes, corresponding to the four roles a participant to the protocol may have. These classes are built by extending entity classes from EVA. The Acceptor and Coordinator classes extend the EVA *TimedConsumerSupplierEntity* class, as both entities are consumers and producers of events and both have periodical tasks (for retransmission of *State* messages and verifying the leader statute). An internal proposer learner, *PLint* is an entity that consumes events and also produces them. Thus, the class implementing the *PLint* extends the *ConsumerSupplierEntity* class defined in EVA. Similarly, we construct the class that defines the behavior of the external proposer learner $PLext$. These entities do not require periodical tasks.

The Paxos-MIC entities define handler methods (one per consumed event type) that implement a code that is very close to the specification of the protocol. These four main entities also define extra data fields and operations required by the protocol. We made the choice of considering three different groups: acceptors, coordinator and external proposers/learners. This choice facilitates the communication exchanges between entities as most of the messages have multiple destinations and can be modeled as multicast communication. Each of the three groups is identified by a group id and has a pair of network multicast address and port associated to it. These network parameters allow inter-groups communication. Each process belonging to a group is uniquely identified by a pair of group id and a process id, which is unique at the level of the group.

In addition to these four main entities, the protocol also relies upon a leader election mechanism. This is provided by a failure detector module queried by acceptors. The easiest way of implementing this mechanism is by defining a *TimedConsumerSupplierEntity* that manages a list of non-crashed coordinators. The entity is in charge of monitoring the events produced by coordinators and based on a timeout mechanism, it simply decides if a coordinator is alive or not. When it is queried by an acceptor interested in knowing the identity of the current leader, the failure detector provides the coordinator with the lowest process identifier among the list of not suspected coordinators.

The second step of the implementation concerns the communication layer. In Paxos-MIC, entities communicate by asynchronously sending and receiving messages which we model through EVA events. We map each type of message exchanged between entities into a type of event. Events that are to be transmitted between entities located remotely are

implemented by defining a new class that inherits from the *RemoteEvent* class. This is the case of *Operation*, *State*, *Decide*, *RetrieveDec*, *Propose* events. In this case, apart from any extra data fields and operations required, the derived class must also provide implementations for the marshall and unmarshall operations necessary to correctly transmit data across different execution platforms. The protocol also defines events that extend the *LocalEvent* class, for modeling messages that are sent locally. For example, the interaction between an acceptor and its associated failure detector module is ensured by local events for monitoring processes activity and triggering failure suspicions.

The synchronous interaction between a coordinator and its associated *PLint* is supplied by EVA services. As a *PLint* is constructed as an extension to a coordinator, operations such as *ProposePull* and *DecidePush* are invoked locally and synchronously. Thus, the services concept defined in EVA is suitable to implement locally invoked operations.

After defining all entities, events and services, the final step in the design is to define how the entities must be connected together. This is achieved by subordinating the defined entities to the appropriate component entities and furthermore, structuring the architecture of the protocol into super-components. This step includes establishing how the entities executed within a process will interact with those executing within another process. This in turn requires identifying the appropriate listener and notifier entities that will allow the cooperation of the different processes that implement the protocol. We made the following choice for designing components: the Paxos-MIC protocol will be executed on nodes (processors) of a grid. On each of these nodes, we deployed various configurations of the core. Thus, on a given node we may have several acceptors, coordinators and *PLext* running. As explained in section 4.1, acceptors and coordinators are part of the core, while external clients are outside this core. This is mainly the reason for creating two separate components for the core members and for outside clients. The core component is in fact a super-component, as it is composed of three other components: a coordinator/acceptor component (that subordinates all acceptor and coordinator entities located on the node), a listener component (that comprises all network listeners) and a notifier component (consisting of all network notifiers). The two last components are responsible for the inter-nodes communication.

5.2.2 Example: the Acceptor class

This subsection presents an extract from the Java class that implements an acceptor. The code defines the handler for the consumption of *Operation* messages sent by the leader. In addition, each acceptor executes a periodical task, in charge of producing a *StateEvent* that notifies the leader of the current state of the acceptor. The example also depicts the extension to the acceptor's code, enabling the entity to handle the receipt of a proposal value from a *PLext*. This extension allows the acceptor to adopt directly the value of the proposer, when optimization $\mathbf{R_O}$ is activated (after the receipt of a *TOP* value from the current leader). The example represents the "translation" in Java of the pseudo-code depicted in Figure 4.3.

```
1
2 public class Acceptor extends TimedConsumerSupplierEntity
3 {
4         public synchronized void operationHandler(OperationEvent opEvent)
5         {
6                 long c;
```

```
7               Tag tag = opEvent.getTag();
8
9               produceMonitor(opEvent.getSrcGid(), opEvent.getSrcPid());
10
11              if ((tag.getRnd() >= this.rnd) &&
12                              (vTag.compareTo(tag) == -1) &&
13                              (opEvent.getVal().getClass().getName().
14                              contains("Bottom") == false)) {
15                      this.vTag.setTag(tag);
16                      this.vVal = opEvent.getVal();
17                      this.initiator = opEvent.getSrcPid();
18              }
19              if (opEvent.getRnd() > rnd) {
20                      rnd = opEvent.getRnd();
21                      this.initiator = opEvent.getSrcPid();
22              }
23              c = tag.getCon();
24              logDVal.put(new Long(c-1), opEvent.getDVal());
25              opEvent.returnShareableEvent();
26              this.produceState();
27      }
28
29      private void produceState()
30      {
31              StateEvent stEvent;
32
33              stEvent = (StateEvent) this.newEvent(StateEvent.class);
34              if (this.vTag.getCon() == 0) {
35                      try {
36                              stEvent.setState(lid, initiator, rnd, vTag,
37                              new Bottom(), new Bottom(), gid, pid,
38                              InetAddress.getByName(this.ip.getHostAddress()));
39                      } catch (UnknownHostException e) {
40                              e.printStackTrace();
41                      }
42              }
43
44              else {
45                      if (vVal.getClass().getName().contains("Top")) {
46                              if (proposals.containsKey(vTag.getCon())) {
47                                      vVal = proposals.get(vTag.getCon());
48                                      proposals.remove(vTag.getCon());
49                              }
50                      }
51                      try {
52                              stEvent.setState(lid, initiator, rnd, vTag, vVal,
53                              logDVal.get(vTag.getCon()-1), gid, pid,
54                              InetAddress.getByName(this.ip.getHostAddress()));
55                      } catch (UnknownHostException e)
56                      {
57                              e.printStackTrace();
58                      }
59              }
```

```
60                this.produceEvent(stEvent);
61        }
62
63
64        public synchronized void proposalHandler(ProposeEventA proposal)
65                throws IncreasingReferenceForNotAllocatedObjectException,
66                InvocationServiceException
67        {
68                boolean already = false;
69                Object val = proposal.getProposal().getVal();
70                String s = val.getClass().getName();
71                long con = proposal.getProposal().getCon();
72                Random r = new Random();
73
74                if ((vVal.getClass().getName().contains("Top"))) {
75                        if ((!s.contains("Bottom")) && (!s.contains("Top"))) {
76                                if ((vTag.getCon() == con) && (rnd == vTag.getRnd())) {
77                                        vTag.setAny(1);
78                                        vVal = proposal.getProposal().getVal();
79                                        produceState();
80                                }
81                        }
82                }
83                proposal.returnShareableEvent();
84        }
85 }
```

5.3 Final words

The Paxos-MIC implementation was written from scratch in Java by relying on an event-driven architecture, that allows to model high-level distributed protocols relying on asynchronous communication. The implementation of the framework defines the classes, corresponding to the four roles a participant to the protocol may have: acceptor, coordinator (and its associated *PLint*) and external proposer/learner, *PLext*. The communication between the entities is modeled through the event consuming/supplying system provided by EVA and also by defining services for synchronous interaction.

Chapter 6

Evaluation of Paxos-MIC

Contents

THE Paxos-MIC implementation described in chapter 5 is evaluated in this chapter through a series of synthetic benchmarks. These benchmarks consist of specific scenarios that facilitate the study of the protocol's performance and the analysis of its behavior. This chapter first describes the experimental settings, then identifies the parameters that influence the protocol's behavior and presents the experimental results obtained during the evaluation.

In a first step, we consider the behavior of the protocol when either only $\mathbf{S_O}$ or both $\mathbf{S_O}$ and $\mathbf{R_O}$ are used. Our aim is to determine the impact of some contextual factors (size of the core, geographical position of the actors) on the time required to reach a decision. In a second step, this chapter focuses on the risky optimization $\mathbf{R_O}$. This optimization fails when collisions occur: during the same consensus instance, at least two different proposers propose different initial values. In the case of a real application, we address the problem of the specification of the trigger criterion and its tuning.

6.1 Experimental setup: the Grid'5000 platform

Our experiments were carried out on the Grid'5000 [34, 14] experimental platform. Grid'5000 provides to the community of researchers a testbed allowing experiments for distributed and parallel computing research. The infrastructure of Grid'5000 supplies a highly-configurable environment, enabling the users to perform experiments under real-life conditions for all software layers ranging between network protocols up to applications.

The platform of Grid'5000 is geographically distributed on different sites located at different places through the French territory. More than 20 clusters spread over 10 sites (see Figure 6.1) are available and each cluster includes up to 64 computing nodes: Bordeaux, Grenoble, Lille, Lyon, Nancy, Orsay, Reims, Rennes, Sophia-Antipolis and Toulouse. Two foreign sites (Luxembourg and Porto Alegre in Brazil) have recently joined the Grid'5000 project, which now comprises more than 7000 CPU cores.

6.1.1 Infrastructure details

Grid'5000 was developed as a hierarchical infrastructure, where the computing nodes are grouped into clusters and several clusters form a site.

Resources. Grid'5000 is designed as a federation of independent clusters and therefore it consists of a complex hierarchy of heterogeneous physical resources. The resource heterogeneity concerns various levels of the architecture, as detailed below [34, 14] :

Processor: The processor families include AMD Opteron (62%) and Intel Xeon EMT64 (32%), featuring mono-core (38.5%), DualCore (32%), QuadCore (27%), 12-core (2.7%) processors.

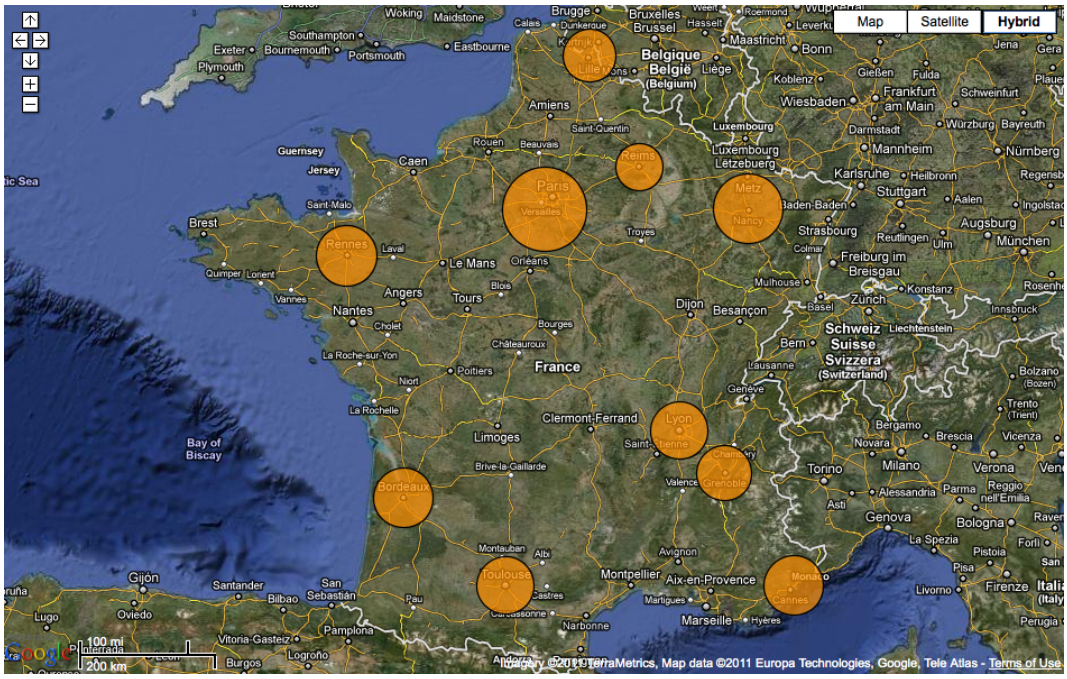


Figure 6.1: Grid sites by number of nodes.

Memory: The nodes are equipped with at least 2 GB of physical memory, which can increase up to 48 GB for some clusters, such as *parapluie* on the Rennes site.

Network: The network interconnects range from Ethernet cards to high-speed Infiniband or Myrinet links for intra-cluster communication.

Although all nodes in Grid'5000 are equipped with Unix-based operating systems, the distributions vary across sites, as well as the set of pre-installed libraries. This issue has to be taken into account for multi-site deployments, as user applications may require specific libraries and tools.

Network. The various sites are interconnected through a high-performance backbone network infrastructure provided by RENATER, the French National Telecommunication Network for Technology, Education and Research. The architecture is based on 10 Gbit/s dark fibers and provides IP transit connectivity, enabling inter-site latencies in the order of 10 milliseconds. Within each site, the resources are typically interconnected through Gigabit Ethernet switches; some sites also provide high speed and low latency interconnects, such as Myrinet or Infiniband, which feature 10 Gb/s and 20 Gb/s links, respectively.

Data Storage. The Grid'5000 architecture provides two levels of data storage:

Local disk: Each computing node is equipped with a local hard disk accessible for user applications. It is however reserved for temporary storage, all data being deleted when the user job is completed.

Shared storage: A network file system (NFS) [56] is deployed on each site, the shared storage server being available from each computing node. It enables users to persistently store data on Grid'5000. This feature is essential for the deployment of distributed applications: the application code and libraries can be installed in the shared storage space and then the user can directly access data and execute the code on multiple computing nodes. Nevertheless, the NFS servers are neither replicated, nor synchronized between Grid'5000 sites. These operations must be manually performed by the user, if a multi-site deployment is required.

6.1.2 Grid'5000 experimental tools

Grid'5000 provides a series of tools that enable users to carry out large-scale experiments and facilitate the deployment of customized environments:

OAR [3] is a batch scheduler that allows Grid'5000 users to make fine-grain reservations, ranging from one processor core to Grid-level reservations spanning over several sites and hundreds of nodes.

Kadeploy [2] enables users to deploy a customized operating system image on the Grid'5000 infrastructure, with administrator rights allowing users to install specific software for their experiments. Moreover, Kadeploy allows users to control the entire software stack and the reproducibility of their experiments.

The Grid'5000 API is a set of well-defined interfaces that enable secure and scalable access to resources in Grid'5000 from any machine through standard HTTP operations.

Taktuk [21] is a tool designed for efficiently managing parallel remote executions on large scale, heterogeneous infrastructures. This tool is particularly useful when the users want to simultaneously start a script on each of the reserved nodes. We used taktuk for each of our experiments in order to deploy the core (acceptors and coordinators) and also to run the clients (external proposers/learners).

6.1.3 Environment settings for Paxos-MIC

We used the OpenJDK 6 Java Virtual Machine with no JIT compilation to run our measurements. Two types of network configurations have been considered for the measurements of the Paxos-MIC protocol. The first one is mono-site (Rennes) inside which the Roundtrip Time (RTT) between two nodes is low (less than 0.140 *ms*). The second one is multi-site as it interconnects different Grid'5000 sites (Orsay, Lille, Toulouse, Grenoble and Rennes). In this configuration, the RTT between two nodes located at two different sites is much higher than within one site and varies according to the considered sites (see Table 6.1).

As explained in Section 5.2, a significant part of the process interaction is modeled as multicast communication. In mono-site configurations, our current implementation of the Paxos-MIC protocol uses multicast listeners/notifiers between nodes located on the same site. However, the routers that connect Grid'5000 sites do not forward multicast traffic. In order to overcome this drawback, for our multi-site configurations, we also implemented UDP relays that transmit multicast packets to nodes located at remote sites. The UDP relays

RTT (in ms)	Lille	Toulouse	Orsay	Rennes
Lille	-	19.8	4.75	10.58
Toulouse	19.8	-	16.51	22.3
Orsay	4.75	16.51	-	9.18
Rennes	10.58	22.3	9.18	-

Table 6.1: Intra-sites RTT

ensure the following: every packet sent over one multicast address is forwarded to the other multicast addresses through several unicasts.

6.2 Overview of the experiments

During the evaluation of the Paxos-MC framework, we aimed at analyzing the performance of the protocol and observe its behavior in several specific scenarios. Our major aim was to identify i) the interest of using either only $\mathbf{S_O}$ or both $\mathbf{S_O}$ and $\mathbf{R_O}$, ii) the main factors that have an impact on the protocol, iii) the cost of $\mathbf{R_O}$ when this optimization is used in bad circumstances, and iv) the potential impact of deploying different roles on remote nodes.

For measuring the performance of the protocol, we selected the classic performance metric for the category of consensus protocols, namely *the latency*. This performance indicator was previously defined in Section 3.7. Recall that the latency is defined by the number of communication steps required by the protocol to reach a decision value, once a proposal is made available, in best-case executions.

A testing scenario is constructed by taking into account parameters that impact the protocol's behavior:

- the occurrence of failures;
- the number of participating entities that defines the size of the core;
- the geographical localization and distribution of the entities on different nodes of Grid'5000;
- the pattern of the consensus sequence; this last parameter characterizes the structure of consensus sequences and is described by:
  - the length of the sequence (or the number of consensus instances executed in sequence);
  - the time between the executions of two consecutive instances (also called *the delay*);
  - the number of proposers;

We carried out most of our experiments in an environment setup consisting of 5 acceptors and 3 coordinators deployed on a local cluster. We also considered that a proposer acts as a learner and we measured the latency of the protocol when executing a sequence of 400 consensus instances. This is what we denote by a classic setup. In the figures displaying

the results, we denote by Paxos-MIC-$\mathbf{S_O}$ the Paxos-MIC protocol executing optimization $\mathbf{S_O}$. Similarly, Paxos-MIC-$\mathbf{S_O}$-$\mathbf{R_O}$ indicates the Paxos-MIC protocol when both optimizations $\mathbf{S_O}$ and $\mathbf{R_O}$ are activated.

6.3 Automatic deployment tools

In order to accommodate the settings of the various testing scenarios, we designed a deployment framework for the Grid'5000 environment. This framework integrates configurable scripts that provide an automatic and fine-tuned deployment and execution of the experiments. The main functions fulfilled by these scripts are the following:

1. **Configure the parameters** of the testing scenarios. The deployment scripts allow the tuning of the parameters that define the testing scenario (the size of the core, number of clients, the length of the consensus sequence, etc.). They generate the configuration files for the core members and the external clients. The entities involved in the execution of the protocol, can be located on the same site or on remote sites, which leads to difficulties in generating the configuration files and in deploying the entities.

2. **Repeatable experiments.** The outcome of the experiments are impacted by extra factors inherent to a distributed testbed (such as, fluctuant network latencies), factors that a user cannot control. Thus, we repeated the experiments several times, under the same conditions, with the purpose of improving the accuracy of the obtained results. The scripts re-execute experiments automatically, for a number of retries specified by the user.

3. **Flexible execution.** The framework should allow users to develop their experiments in a flexible manner, so that they can perform evaluations that do not depend on predefined parameters or system settings.

The deployment and execution framework consists of scripts that operate at three important levels:

Global settings. The scripts also handle the global settings that users must define when employing the Grid'5000 platform. For instance, to use the Grid'5000 platform, one has to specify the type of OAR reservation used and the required identifiers, such as the reservation id and associated key. Moreover, the scripts include the type of connector used to contact the nodes (e.g. ssh), a list of the Grid'5000 sites involved in the deployment and global environment variables. A particular type of script is in charge of propagating the global variables on all the nodes and of synchronizing the configurations on each Grid'5000 site.

Deployment specification. These scripts are in charge of defining the entities that will be executed on specific nodes. These scripts receive as an input, a file containing the names of the Grid'5000 nodes obtained after making a reservation, and they automatically generate the files required to define and start the Paxos-MIC entities. The description of the entities comprises configuration files containing specific parameters and requirements. For example, for each coordinator or acceptor, the configuration file

must "know" the IP address of the Grid'5000 node on which the entity will be run. In addition, multicast addresses and ports are included in these configuration files, being mandatory for ensuring multicast communication among the entities of Paxos-MIC.

Results handling. To collect all the logs generated by the deployed platforms, our framework takes advantage of the shared-user directory enabled by the NFS on Grid'5000. Dedicated scripts gather all the nodes logs and parse the files in order to obtain relevant results. The numbers are passed as an input to scripts that output a graphical representation of the results.

6.4 Results

In this section, we describe the experimental evaluation and discuss the obtained results. With each testing scenario, we gradually varied each of the aforementioned parameters and observed its impact on the protocol's behavior.

6.4.1 Failures

In this series of experiments, we evaluate the impact of failures on the Paxos-MIC protocol. As explained in Section 4.3.2, the Paxos-MIC protocol has been designed to tolerate up to f_c crashes of coordinators and up to f_a crashes of acceptors.

In terms of performance, the failure of an acceptor has no effect when all acceptors are located on a single cluster. Conversely, when the acceptors are deployed over several sites, the failure of an acceptor near the leader (in terms of communication delay) is detrimental to all future consensus instances. To be convinced, the reader is invited to look at the Table 6.2 that identifies variations in cost depending on the positioning of the acceptors. The failure of acceptors that are far from the leader (in terms of communication delay) is not problematic.

Obviously, the failure of a coordinator that is not acting as a leader has no impact. The failure of a leader creates an important additional cost which depends essentially on the time required by the leader election mechanism to suggest a new correct leader. Although this time is very high, its analysis is not necessarily very informative as the cost is proportional (in our implemented solution) to the timeout values used to detect failures. It seems more interesting to focus on the additional cost caused at the level of Paxos-MIC by the arrival of a new leader.

More precisely, in Figure 6.2, we show the latency observed when 400 consensus are executed by the same leader during the same *Propose* phase and the latency observed when each consensus requires the execution of both a *Prepare* phase and a *Propose* phase. As a two-phase execution is required each time a new leader appears, this second measure indicates the (small) additional cost induced by the Paxos-MIC protocol itself.

In this experimentation, both the number of acceptors and the number of coordinators increase: $n_a = 2f + 1$ and $n_c = f + 1$, where f varies from 1 to 15. Thus, in a first step we have a core of 2 coordinators and 3 acceptors, which gradually increases up to 16 coordinators and 31 acceptors. The core members are deployed on a local cluster. A client behaves both as a proposer as well as a learner and it is in charge of proposing a sequence of 400 initial values, for each of the core configurations. We measure in each configuration,

the latency of Paxos-MIC when no failures occur and the latency of Paxos-MIC when an additional cost is required for executing a new *Prepare* phase for each consensus instance.

The experiment described above is performed for each of the core configurations, which results in the curves depicted in Figure 6.2. We observe that for a number of acceptors $n_a < 12$, the two curves evolve similarly: if the leader changes, an additional 30 % cost is observed. Beyond 13 acceptors, the gap between the two curves increases and goes up to an additional cost of 50 %. In fact, the execution of the two phases requires two successive observations of a majority quorum (one at the end of each phase). When the number of acceptors increases, the number of messages that have to be gathered in order to achieve a quorum also increases.

6.4.2 Scalability

One of our goals was to assess how the protocol scales when the number of coordinators and acceptors increases. This particular parameter, namely the size of the core, also allows to determine the interest of the optimization $\mathbf{R_O}$.

We consider an experimental setup that is favorable to this optimization. A single *PLext* is involved in the 400 consensus. 50 ms elapses between the receipt of a decision value related to a consensus c and the sending of an initial value for the consensus $c+1$. Both the number of acceptors and the number of coordinators are increased: $n_a = 2f+1$ and $n_c = f+1$. We deployed the core members on the clusters of the Orsay sites. Our goal is to see how the protocol is scalable when the number of acceptors increases up to 51. For each of the possible core sizes, we ran 400 consensus instances and measure the latency of the protocol with its two behaviors: Paxos-MIC-$\mathbf{S_O}$ and Paxos-MIC-$\mathbf{S_O}$-$\mathbf{R_O}$. Figure 6.3

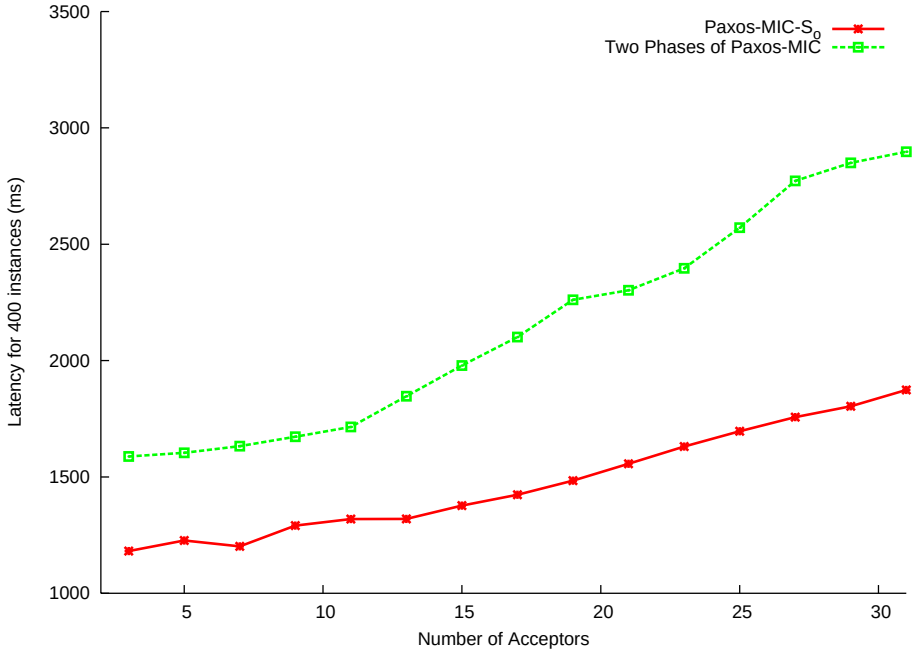


Figure 6.2: Cost of the Prepare phase (in case of a failure).

displays the obtained results.

When the number of acceptors is relatively low (below 15), optimization $\mathbf{R_O}$ allows gains ranging from 30% to 50%. When the number of acceptors increases, the interest of the optimization $\mathbf{R_O}$ decreases. The two curves converge when the number of acceptors is equal to 37. At this point the cost of an *Any* quorum (gathering of at least 28 *State* messages) is much greater than the cost of a majority quorum (gathering of at least 19 *State* messages). The gain obtained by bypassing the leader is no longer sufficient to hide the growing additional costs of an *Any* quorum. However, up to 50 acceptors, the worst performance with optimization $\mathbf{R_O}$ (in favorable circumstances) is never higher than 4%. Note that the slope followed by the two curves changes slightly when 35-40 acceptors are used. At this stage, all the acceptors were no longer located on the same cluster. This has an impact on the measurements. For this particular experimental settings, we always ensure that every node implements at most one role (coordinator, acceptors, external proposer/learner). Thus, we used up to 80 nodes.

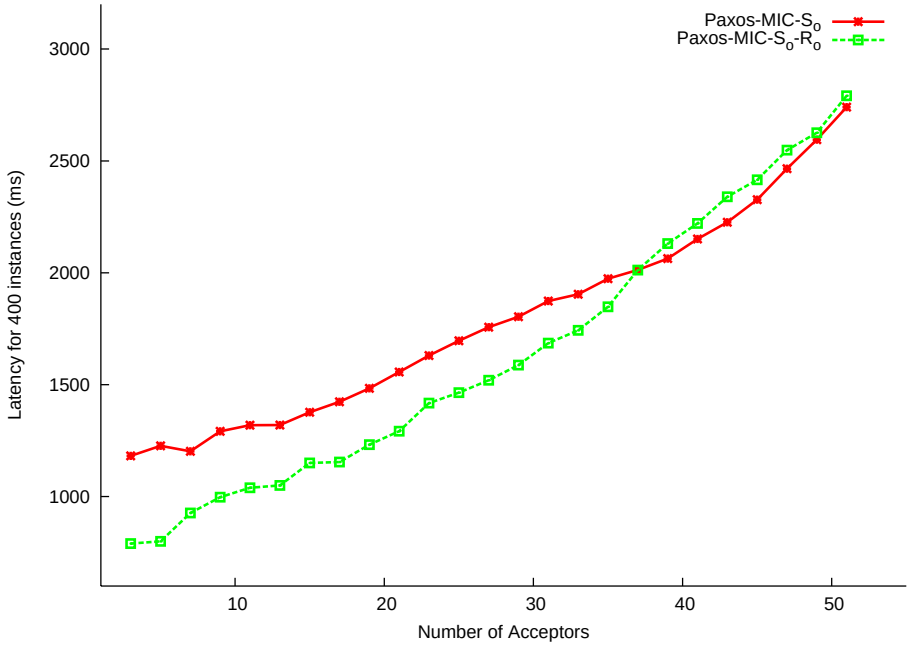


Figure 6.3: Scalability.

As a concluding remark, we observe that a higher number of acceptors (up to 51) impacts optimization $\mathbf{R_O}$ to a greater extent than optimization $\mathbf{S_O}$, as $\mathbf{R_O}$ requires greater quorums (*Any* quorums as opposed to classic majority quorums). However, up to 50 acceptors, the worst degradation of $\mathbf{R_O}$ (in favorable circumstances) is never higher than 4%.

6.4.3 Localization

This subsection studies another parameter that influences the protocol's latency, namely the localization of the entities. By considering different distributions of the entities on different sites, we aim at observing the impact of entities localization on the performances. For this

purpose, we deployed 3 coordinators, 5 acceptors and one proposer on four different sites of Grid'5000: Rennes, Lille, Orsay and Toulouse. In the Table 6.2, deployment scenarios are denoted by D_i.

In both scenarios D_1 and D_2, the proposer is co-located with the leader and a majority of acceptors. The leader C_l has direct access to a local majority quorum composed of acceptors located in Rennes. However, when optimization $\mathbf{R_O}$ is used, the *Any* quorum has a cardinality of 4. When all the acceptors are no longer co-located (case D_2), the latency increases considerably, as one of the acceptors located in Lille is necessary to gather an *Any* quorum.

Consider cases D_3 and D_4: a majority of acceptors is deployed on a different site than the leader and the proposer, which are co-located. By spreading acceptors over 3 different sites, we observe a slight variation of the latencies for both optimizations. However, as the links Rennes-Lille and Rennes-Orsay have similar network delays, the two optimizations are slightly impacted. The best scenario for $\mathbf{R_O}$ requires that all entities are co-located. In this way, the proposer is able to rapidly reach both the leader and the acceptors.

Scenario D_5 places the proposer remotely from both the leader and the acceptors. By deploying the proposer in Toulouse, both optimizations degrade considerably as the link delay between Toulouse and Rennes is very high. Scenarios from D_6 to D_8 deploy different distributions of acceptors over the 4 sites. When the proposer moves from Orsay (case D_{6a}) to Toulouse (case D_{6b}), the costs of both optimizations increase considerably, due to the links between Toulouse and every other site, which are much more cost-full. Again, we notice that when optimization $\mathbf{R_O}$ is used, the protocol hardly reacts to changes in acceptors topology.

	Rennes	Lille	Orsay	Toulouse	Paxos-MIC+$\mathbf{S_O}$	Paxos-MIC+$\mathbf{R_O}$
D_1	P C_l C C A A A A A				1275 ms	859 ms
D_2	P C_l C C A A A	A A			1273 ms	5147 ms
D_3	P C_l C C	A A A A A			5853 ms	5417 ms
D_4	P C_l C C	A A	A A	A	5888 ms	5499 ms
D_5	C_l C C A A A A A			P	10276 ms	10078 ms
D_{6a}	C_l C C	A A A	P A A		8929 ms	6099 ms
D_{6b}	C_l C C	A A A		P A A	15239 ms	11983 ms
D_{7a}	C_l A	C A A	C A A	P	13626 ms	11922 ms
D_{7b}	C_l A A	C A A	C A	P	13630 ms	11967 ms
D_8	C_l	C A A A	C A A	P	14851 ms	12465 ms

Table 6.2: Localization

From the obtained results, we can conclude that the best scenario for $\mathbf{R_O}$ requires that all entities are co-located. In this way, the proposer is able to rapidly reach both the leader and the acceptors. As soon as one acceptor relocates on a different site, optimization $\mathbf{R_O}$ degrades in latency, while optimization $\mathbf{S_O}$ is less impacted by this change. By spreading acceptors over 3 different sites, we observe a slight variation of the latencies for both optimizations.

6.4.4 Delays

The purpose of this series of experiments is to assess how the protocol behaves in more "stressful" conditions. The parameter that allows such an assessment is the delay between two consecutive consensus instances.

We consider the deployment of 5 acceptors and 3 coordinators on a local cluster on the site of Rennes. A single external proposer/learner located on the same cluster proposes initial values at regular time intervals. We vary this time interval from 0 to 50 ms, the value is gradually increased by a step size of 5 ms. At each step, we ran 400 consensus instances and measured the total latency for such a sequence. Thus, we obtained the curves presented in Figure 6.4.

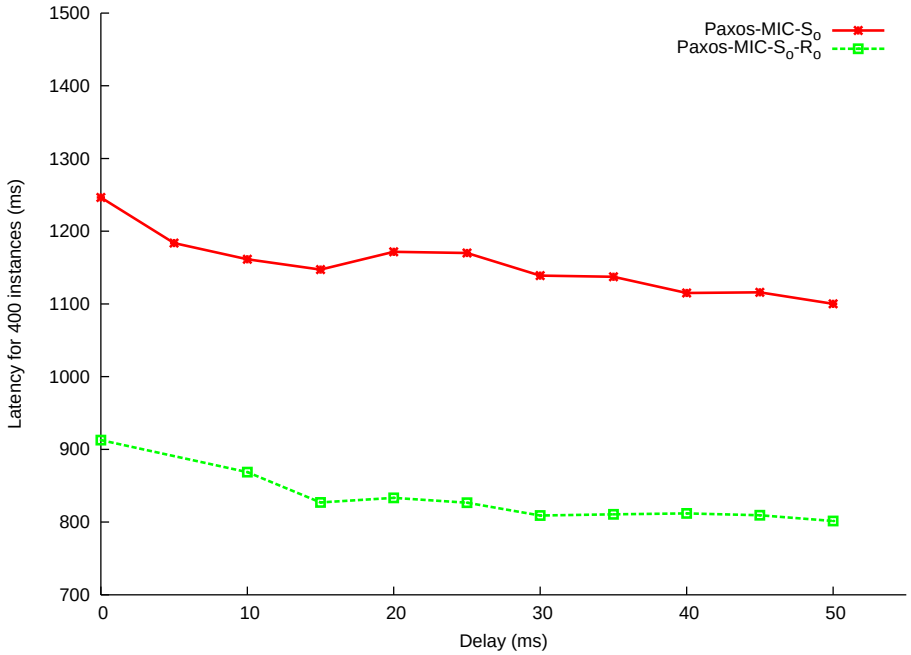


Figure 6.4: Delays.

If we analyze the results, we observe that beyond 15 ms, the cost remains relatively stable. For smaller values, a rapid succession of consensus leads to an increase in latency. Yet this increase is limited (about 10%). As we have a single external proposer/learner, optimization $\mathbf{R_O}$ was always successful. The interest of optimization $\mathbf{R_O}$ appears clearly in these favorable circumstances: the gain is greater than 20%.

6.4.5 Participation to several consensus instances

The usual pattern of a consensus sequence is the following: once an external proposer/learner obtains the decision for consensus c, it immediately provides an initial value for the following consensus, $c+1$. Paxos-MIC allows a degree of parallelism at the higher-level of the external proposer/learner. More precisely, the internal proposer/learner *PLint* manages the interface between the protocol itself and the external proposers. The behavior of the *PLint*, as described in Section 4.3.2, allows this small degree of parallelism. Although

the consensus instances are still executed sequentially, we show that there is an interest in allowing an external proposer/learner to be involved in several consensus instances at the same time. However, the gain is obtained only if an external proposer participates to two consensus instances at the same time.

To estimate the potential benefit that may be provided by this additional degree of freedom, we have executed the protocol Paxos-MIC-$\mathbf{S_O}$ in two different settings. We deployed 3 coordinators and 5 acceptors on a cluster located in Rennes. A single external proposer/learner (denoted PL_{ext_1} and also located in Rennes) was using the agreement protocol. In the first case, PL_{ext_1} never participates to more than one consensus instance. Once it gets the decision corresponding to the consensus c, it immediately provides an initial value for the following consensus $c+1$. In the second case, PL_{ext_1} participates simultaneously to two consecutive consensus instances. When it receives the decision corresponding to the consensus c, it calls the function $Propose(c+2, v)$. In that case, the total cost corresponding to the execution of a series of 400 consensus decreases by 29% (See Table 6.3). Indeed, in the second case, the sending of all initial values from PL_{ext_1} to the leader (except the first) and the sending of all decision values from the leader to PL_{ext_1} (except the last) are done while consensus are in progress. Of course, this observed benefit can actually be obtained only if an application is able to progress alternately in the construction of two sequences of independent decisions (for example, Atomic Broadcast).

One consensus at a time	Two consensus at a time	Three consensus at a time
1226.902 ms	871.282 ms	872.386 ms

Table 6.3: A single proposer's participation to 400 consensus.

Executing consensus instances in parallel in other works: A higher degree of parallelism is provided in [8]. The protocol described in [8] provides a modular deconstruction of the Paxos protocol [38]. The authors identify building blocks of Paxos, denoted by *abstraction*, such as *weak leader election* and *round based consensus*. The latter acts as a sub-protocol used to agree on the decision value and it relies upon a *round based register* to store and lock the decision value. The register plays the role of an acceptor in the classic Paxos protocol [38].

Among other numerous contributions, the authors show in [8], how the deconstruction of Paxos into independent abstractions can be used in implementing higher level applications. They consider the *Total Order Broadcast and Delivery* protocol. Whenever a process wants to broadcast a message, it sends the message to the current leader. The leader triggers a new consensus instance by proposing a batch of messages composed of all the messages received so far and not yet delivered. In their approach, a consensus instance corresponds to a single instance of total order, more precisely to a single batch of messages. Each new consensus instance creates a new register instance, thus leading to an array of round-based registers (each register stores and locks the decision value for a given consensus instance). The round-based register is used for logging the decision value. Any correct process must be able to retrieve the decision value for any completed consensus instance. For achieving this purpose, the register instances must remain active even after the corresponding consensus instance has finished. The logged information is available and can be retrieved at any time. Such a logging mechanism requires that (possibly) a high number of register instances are kept active.

Several consensus instances are executed in parallel due to the "one to one" association between a consensus instance and a register instance. This parallelism brings the advantage of a low latency for reaching decisions. However, making us of a decision is a time-consuming process. Let us assume that k consensus instances were launched and executed simultaneously. These instances correspond to k batches of messages whose intersection is a non-empty set. When a process p_i receives the decision for a consensus instance k, it first verifies that the received decision is the next decision that was expected. p_i will deliver messages in batch k, only if it has previously delivered batch $k-1$. When delivering the batch of messages number k, p_i must deliver only the messages that were not yet delivered: for the $k'th$ new decision (more precisely, batch of messages), p_i must filter a window of $k-1$ previous decisions and deliver only the messages contained in batch k and not yet delivered when consuming all previous $k-1$ decisions.

6.4.6 Collisions

The optimal conditions in which optimization $\mathbf{R_O}$ has a positive impact on the latency by allowing a performance gain, require the absence of collisions. In performing our experiments, we also focused on assessing the performance degradation when the conditions that have motivated the use of optimization $\mathbf{R_O}$ become less favorable.

We considered a simple scenario: we deployed 3 coordinators, 5 acceptors and 2 proposers on 10 nodes of a local cluster situated on the Rennes site. We analyze the two possible behaviors of Paxos-MIC-$\mathbf{S_O}$ and Paxos-MIC-$\mathbf{S_O}$-$\mathbf{R_O}$. We also slightly modify Paxos-MIC to execute a new *Prepare* phase after each decision. In that case, the observed behavior is quite similar to that of classic Paxos (with no optimization).

In order to simulate the degradation of the conditions favorable for optimization $\mathbf{R_O}$, we introduce the notion of *probability*. As we have 2 proposers, p is defined as the probability that an acceptor adopts the value from the first proposer while (1-p) is the probability to adopt the value from the second one. Figure 6.5 illustrates the latency (y-axis) measured for the three behaviors of Paxos-MIC, previously mentioned. For each point on a curve, we run 400 consensus instances and we compute the mean latency for one instance. The parameter that varies in this set of experiments, is the probability p, previously defined. On the x-axis, the value of p varies from 1 to 0.5. When p = 1, no collisions occur as the value provided by the first proposer will always be adopted by all acceptors. The worst scenario happens when both values have equal probabilities of being adopted by acceptors, in which case p = 0.5.

The probability does not impact optimization $\mathbf{S_O}$ nor the behavior of Paxos-MIC when no optimization was used. However, the latencies of these two behaviors (which are constant) are also displayed in the figure, to serve as a comparison point.

When p = 1, we measured for Paxos-MIC with both optimizations a latency that is more than twice better than the one obtained for the Classic Paxos protocol. Compared to the Paxos-MIC-$\mathbf{S_O}$, Paxos-MIC with both optimizations exhibits a significant gain as latency is reduced by 30%. As expected, the latency of the Paxos-MIC with both optimizations starts to be higher than the latency of the Paxos-MIC-$\mathbf{S_O}$ protocol very quickly (when $p < 0.97$). In other words, even a small number of collisions is sufficient to eliminate the performance gain brought by optimization $\mathbf{R_O}$. Then, when p reaches almost 0.7, the latency of Paxos-MIC with both optimizations is higher than the latency of Classic Paxos. However, we can

observe that even higher, the latency of Paxos-MIC-$\mathbf{S_O}$-$\mathbf{R_O}$ stays reasonable as it never excesses a 15% penalty in the worst case (p=0.5).

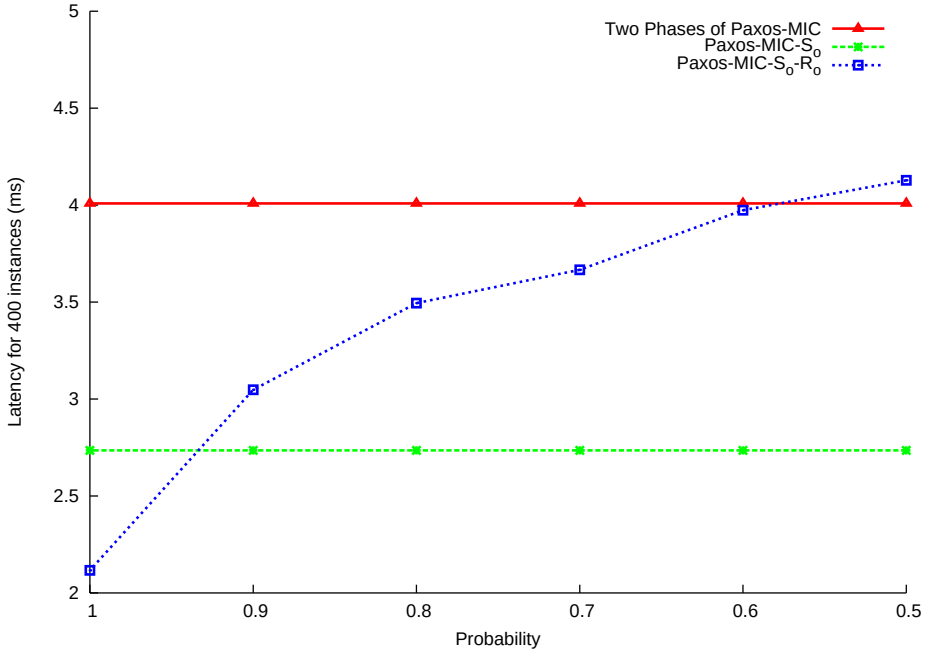


Figure 6.5: Probability of collisions (two proposers).

We now extend the previous scenario by considering more than one collision and analyzing the degradation of the latency required by optimization $\mathbf{R_O}$. We vary the number of colliding proposals from 1 to 9 and we represent in Figure 6.6 the latency for 400 consensus instances. For a given number x of simultaneous proposals, we consider the worst case: all values have equal chances of being selected by acceptors (the probability for each value is p = $\frac{1}{x}$). The results show that as soon as a collision appears, the latency increases considerably as its value becomes twice as great. By increasing the number of colliding proposals, the chances of a leader gathering the same value from an *Any* quorum of acceptors, decrease and thus impacting the latency of optimization $\mathbf{R_O}$. However, the degradation is still reasonable even if the number of collisions increases.

In the first scenario of this subsection, we considered only two proposers that may collide while providing their values. Further, we extended this scenario by considering several collisions. Our results show that optimization $\mathbf{R_O}$ is impacted considerably when a collision occurs (the number of simultaneous proposals = 2) but degrades reasonably once the number of collisions increases.

6.5 Zoom on $\mathbf{R_O}$ - Prediction of collisions

As explained in Section 4.4, the $\mathbf{S_O}$ optimization reduces the latency to four communication steps. When both $\mathbf{S_O}$ and $\mathbf{R_O}$ are combined, the cost is reduced to three in favorable circumstances. However, as the optimization $\mathbf{R_O}$ is unsafe, its use can be counterproductive. One of the main benefits of Paxos-MIC lies in the fact that it allows to determine if the $\mathbf{R_O}$

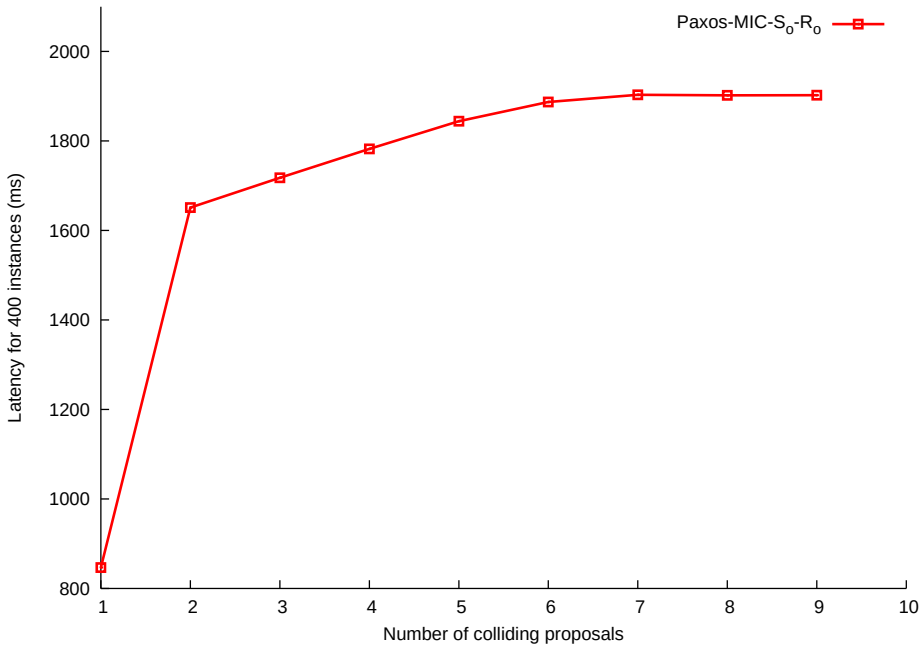


Figure 6.6: Collisions (several proposers).

optimization has to be triggered or not at runtime. This choice is made by the leader between two consecutive consensus instances if an idle period occurs. The test evaluated by the leader is called *a triggering criterion*: its evaluation is done locally and does not require any synchronization with other actors.

This section zooms on the risky optimization $\mathbf{R_O}$. This optimization fails when collisions occur: during the same consensus instance, at least two different proposers propose different initial values. So far in this chapter, we analyzed the behavior of Paxos-MIC in different scenarios. This analysis provides a better understanding of the protocol. Furthermore, it leads us to define four reference contexts that are characterized by the knowledge of three durations, namely the time required to reach a decision i) when $\mathbf{R_O}$ is not triggered, ii) when $\mathbf{R_O}$ is successfully triggered, and iii) when $\mathbf{R_O}$ is triggered in bad circumstances. In a second step, we propose several triggering criteria. Some are static (*Always*, *Never*) while others are dynamic (*Time*, *Result*, *Random*). To compare them, we consider a particular application (a secure Web server) and a real trace that records the activation dates of the successive consensus instances during a period of 16 days. By simulation, we evaluate the expected gain in each of the four contexts. Even if they fail to predict accurately the future collisions, some triggering criteria allow nevertheless to adapt more or less the behavior of the protocol to the current situation.

6.5.1 Four main reference contexts

Based on the above observations, we identify four different contexts, in which we vary two important factors: the core size and the localization. In two of them (C_{L5} and C_{W5}), the core is composed of 3 coordinators and 5 acceptors. In the two others, the core is composed of 6

coordinators and 11 acceptors. In two of them (C_{L5} and C_{L11}), the whole core is on the same site (Rennes). In the two others, external proposers-learners are on a same site, coordinators and a minority of acceptors are on a second site while a majority of acceptors are located on a third site. Table I.2 indicates three durations of consensus that have been observed during experimentations: D_{succ} ($\mathbf{R_O}$ is triggered and no collision occurs), D_{norm} ($\mathbf{R_O}$ is not triggered), and D_{fail} ($\mathbf{R_O}$ is triggered and a collision occurs). In the four contexts, the optimization $\mathbf{R_O}$ may be of interest if $D_{succ} < D_{norm} < D_{fail}$. The ratio is defined as (D_{fail} - D_{norm}) / (D_{norm} - D_{succ}). It indicates the number of success that are required to compensate a failure.

	D_{succ}	D_{norm}	D_{fail}	ratio (fail/succ)
Context C_{L5}	1.31ms	1.92ms	3.15ms	2.0
Context C_{L11}	1.78ms	2.12ms	5.18ms	8.8
Context C_{W5}	18.68ms	25.88ms	42.20ms	2.3
Context C_{W11}	21.59ms	26.01ms	42.82ms	3.8

Table 6.4: Mean Duration of a Consensus.

6.5.2 How collisions occur

Optimization $\mathbf{R_O}$ improves the latency of the protocol only when the circumstances are favorable, *i.e.* when no collision occurs. Here, we analyze the performance degradation when the conditions become less favorable. As mentioned before, a collision can occur only if several different values are proposed during a same consensus instance. A simple scenario is considered: 3 coordinators, 5 acceptors and 2 proposers P_a and P_b are deployed on 10 nodes of a local cluster (context C_{L5}). The proposer P_a provides a value v_a while P_b proposes always the value v_b. Let us consider that each acceptor has a probability p to adopt the initial value v_a and a probability $(1-p)$ to adopt the initial value v_b. The leader has to execute the recovery procedure (i.e., the optimization $\mathbf{R_O}$ fails) if both the value v_a and the value v_b appear among the four values gathered by the leader. Let us consider the set of four gathered values. Four values v_a can be collected with a probability p^4. Four values v_b can be collected with a probability $(1-p)^4$. Thus, the probability for the optimization $\mathbf{R_O}$ to succeed is equal to $p^4 + (1-p)^4$. Consequently, if we consider a sequence of consensus instances (all executed with the optimization $\mathbf{R_O}$ and with an important delay between each pair of consecutive instance), the average duration $D_{average}$ of a consensus can be approximated by the following formula: $(p^4 + (1-p)^4)D_{succ} + (1 - p^4 - (1-p)^4)D_{fail}$. There is a gain if $D_{average} > D_{norm}$. Thus, in context C_{L5}, using $\mathbf{R_O}$ is of interest if $p > 0.9$. In other words, the optimization $\mathbf{R_O}$ is not risky if one of the two values has about 9 times more chance to be adopted by each acceptor. In the worst case, both values v_a and v_b have the same probability to be adopted: $p = 1 - p = 0.5$. In that case, an additional cost of 52% compared to the normal duration is observed.

Assume that S_x (with $x \in 0, 1, 2$) denotes the following property "one of the two values (v_a or v_b) is adopted by exactly x acceptors". When the 5 acceptors are correct, only one property (S_0, S_1, or S_2) holds while the two others are false. If property S_0 is satisfied, no collision can occur. If property S_2, holds a collision is unavoidable: the 4 values gathered by the leader cannot be all equal. When S_1 is satisfied, the probability that a collision occurs is equal to $4/5$ if all communication channels are fair. To verify the above theoretical results

during experiments, we have artificially controlled the probability that an acceptor adopts v_a (or v_b). All the above formula are satisfied when we launch 400 consensus instances. The conditions that lead to observe either the property S_0, S_1, or S_2 are difficult to master. Of course, the fact that the two proposers act quite simultaneously is the main requirement. Yet, other factors (traffic intensity, broadcast mechanism, local or wide area network, ...) may increase or reduce the risk.

6.6 Simulation of Paxos-MIC

As one of the main interests of Paxos-MIC resides in the ability to trigger $\mathbf{R_O}$ when it is appropriate (and not to trigger it when it is not appropriate), we focus now on the definition of triggering criteria for $\mathbf{R_O}$. In this section, we aim at assessing various triggering criteria for $\mathbf{R_O}$ in the context of a real application that relies upon the consensus in order to progress. The application is a replicated Web architecture in which HTTP requests have to be ordered, thanks to the Paxos-MIC protocol, before being sent to the Web servers. We use a log of HTTP requests and, for each request, we simulate an execution of Paxos-MIC. Four different configuration contexts (see Section 6.5.1) are retained and for each of these contexts, we evaluate the quality of the proposed triggering criteria for $\mathbf{R_O}$.

6.6.1 Application: A secure Web Architecture

In the context of a project called DADDI, we have designed an architecture which provides an IDS (Intrusion Detection System) component in charge of detecting intrusions in an information system by comparing the outputs delivered by several diverse Web servers [31, 59]. The architecture (see Figure 6.7) is organized as follows. A proxy handles the clients' requests. It forwards the requests received from a client to the Web servers and later forwards the response received from the IDS to this client. It ensures that the Web servers receive the same sequence of requests and thus evolve consistently.

In this approach, the idea is to take advantage of the existing software and hardware diversity: as the Web servers have been designed and developed independently, they do not exhibit the same vulnerabilities. If we assume that a malicious payload contained in a Web request cannot take advantage of two different vulnerabilities, then an intrusion may occur in only one Web server at a time. In this case, as the other Web servers are not exhibiting the same vulnerability, they are not affected by the attack and they all provide a same response that is supposed to be different from the response provided by the corrupted Web server. The IDS is in charge of comparing the responses returned by the servers. To select the response that has to be sent back to the client, it uses a majority voting algorithm. If it detects some differences among the responses, it raises an alarm. We showed in [31] that this solution guarantees confidentiality and integrity.

Availability is partially offered at the server level thanks to the set of diverse Web servers. To enhance availability, it is also necessary to replicate the reverse proxy/IDS and hence to form a group of reverse proxies. To ensure that the group of reverse proxies/IDS evolve consistently, despite the occurrence of failures, an atomic broadcast service has to be used. This atomic broadcast service relies upon a consensus protocol to order the requests.

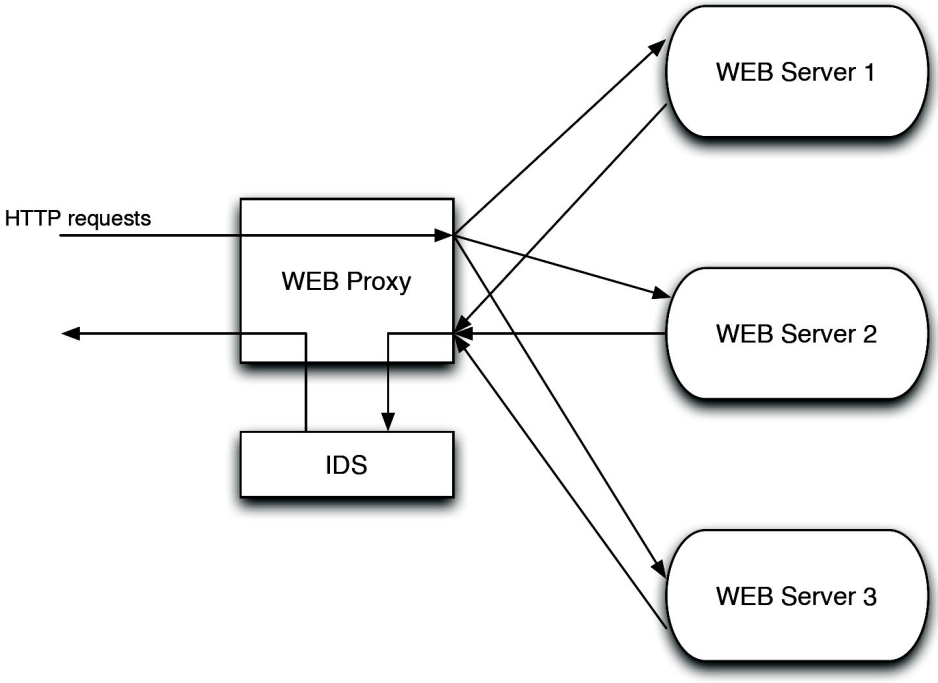


Figure 6.7: Architecture of the secure web-server.

Among the members of the group, a leader is elected. It is in charge of broadcasting the requests to the set of diverse Web servers. In [31], we proposed a solution in which each request to be submitted to the Web servers is only addressed to the leader of the group of reverse proxies. In this particular approach, the leader is the unique member of the group to propose value and hence the $\mathbf{R_O}$ optimization is effective as no conflict could happen. An alternative, considered in the rest of this section, would be to have every reverse proxy receiving requests. Thus, the values are potentially proposed by more than one proposer and they may differ. The interest of triggering the $\mathbf{R_O}$ optimization for the consensus protocol depends on the probability that at least two different requests are "simultaneously" proposed (occurrence of a collision) for a same consensus instance.

6.6.2 Log analysis

We use a log of HTTP requests that has been collected during a period of 16 days. During this period, 573753 requests were addressed to the Web server of a french engineer school. The log contains lines of requests. For each request, the host originating the request, its arrival time (with 1 μs resolution), and the body of the request are available. The frequency of requests is not very high (about 25 requests per minute in average) but this frequency is not uniform on the whole log: the night periods have considerably less requests than the day periods. We found in the log a period of 10 hours that is six times more dense that the whole log. As it is not enough to represent a heavy loaded server, we decided to compute a new log where the time interval between two consecutive requests of the original log is divided by 60. Thus, the average frequency of requests is about 1500 requests by minute in this new log. Hence, we have two logs: the original log and a compacted log on which we can test

our triggering criteria.

Based on the knowledge of the arrival times of the requests, we simulate the execution of the Paxos-MIC protocol for various triggering criteria of $\mathbf{R_O}$ (see Figure 6.8 for the algorithm executed by the simulation). The algorithm loops over the individual requests contained in the log. Two separate parts are identified inside the loop. The first part (lines 6 to 23) of the algorithm is dedicated to the evaluation of success/error in the decision related to the activation of $\mathbf{R_O}$ for the request r_{prev}. The $triggered_{R_O}$ boolean variable indicates if $\mathbf{R_O}$ was triggered during the previous consensus. The *idle* boolean variable indicates if there was an idle period before the previous consensus. Once the arrival time at_{curr} for the current request is known, the ending time of the consensus ordering the request r_{prev} is computed. If $\mathbf{R_O}$ has been triggered for request r_{prev} , the computed ending time depends on the occurrence of a collision. A collision is considered to occur if the difference between at_{curr} and at_{prev} is less than δ. In that way, δ represents in our simulation *the estimation of the risk of a collision*. We choose to fix the value of δ to be equal to D_{succ}. This choice means that a collision would be observed if $\mathbf{R_O}$ was triggered for r_{prev} and the arrival time of r_{curr} falls in the time interval of the consensus ordering r_{prev}. Our goal being to evaluate the interest of $\mathbf{R_O}$ and the accuracy of the triggering tests, we argue that this overestimation does not disclaim our conclusions.

The second part (lines 25 to 34) of the algorithm is related to the decision of activating $\mathbf{R_O}$ (or not) for the current request. If the arrival time of the current request occurs before the ending time of the previous consensus (line 25), the activation is non sense (lines 29 to 30). Indeed, activating $\mathbf{R_O}$ requires the leader to send a special mark $\top$ to the acceptors to force them to directly accept a value from the proposers. If the value is already available, it does not make sense for a leader to send $\top$ instead of the real value. If the activation is possible, the *triggering criteria* that allows to decide if $\mathbf{R_O}$ has to be triggered or not, is executed (line 27). This test is either trivial (static tests like *Always* or *Never* that return the same boolean value, true or false) or more sophisticated (tests based on the knowledge of the past or even on the future for unrealistic tests).

The tree given in Figure 6.9 shows the different percentages of successes and errors that will be used in our evaluation. All can be computed thanks to the algorithm of Figure 6.8. In the BUSY state, a new consensus starts immediately because at least one initial value is available. We call such a consensus, an immediate consensus. The triggering criterion is evaluated only when an idle period occurs.

6.7 Triggering criteria

The ability to decide whether to trigger optimization $\mathbf{R_O}$ is one of the main interests of Paxos-MIC. We implement different triggering criteria and we evaluate their effectiveness. They are classified in three categories: static, dynamic (realistic) and unrealistic, which we use as references for comparison purposes with the other ones.

6.7.1 Classification

1. **Static Criteria** The static tests give always the same results: the *Never* test does not trigger $\mathbf{R_O}$ and the *Always* test triggers $\mathbf{R_O}$, each time they are evaluated (Figure 6.8,

```
(1)  triggered_{R_O}, idle ← True; endingTime ← 0
(2)  N_success1, N_success2, N_error1, N_error2, N_immediate ← 0
(3)  r_prev ← readline (log); at_prev ← timestamp (r)
(4)  While (log not empty) do
(5)      r_curr ← readline (log); at_curr ← timestamp (r)
(6)      // PART 1
(7)      if (triggered_{R_O}) then
(8)          if (at_curr − at_prev) > δ then
(9)              N_success2 ← N_success2 +1
(10)             endingTime ← at_prev + D_succ
(11)         else
(12)             N_error2 ← N_error2 +1
(13)             endingTime ← at_prev + D_fail
(14)     else
(15)         if (!idle) then
(16)             N_immediate ← N_immediate +1
(17)             endingTime ← endingTime + D_norm
(18)         else
(19)             endingTime ← at_prev + D_norm
(20)             if (at_curr − at_prev) < δ then
(21)                 N_success1 ← N_success1 +1
(22)             else
(23)                 N_error1 ← N_error1 +1
(24)     // PART 2
(25)     if (at_curr > endingTime) then
(26)         idle ← True
(27)         test = Evaluate (triggeringtest)
(28)         if (test = true) then
(29)             triggered_{R_O} ← True
(30)         else
(31)             triggered_{R_O} ← False
(32)     else
(33)         triggered_{R_O} ← False
(34)         idle ← False
(35)     r_prev ← r_curr; at_prev ← at_curr
```

Figure 6.8: Simulation of Paxos-MIC.

line 27). These triggering criteria are clearly not optimal: depending on the context and the frequency of the consensus requests, the *Always* test can lead to many collisions degrading the performance of the architecture and the *Never* test will not be able to take advantage of $\mathbf{R_O}$.

2. **Dynamic Criteria** Two of the three dynamic tests that we define use knowledge based on the past to decide whether to trigger or not $\mathbf{R_O}$.

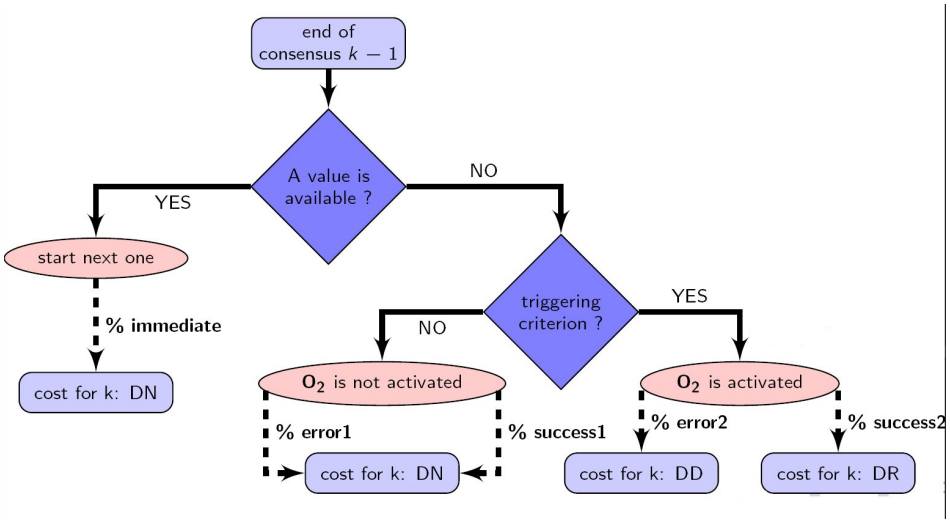


Figure 6.9: Observed durations, percentages of success/error.

- The *Time* test triggers $\mathbf{R_O}$ if there is no consensus request during a period of at least Δ ms after the end of the last consensus (in our simulations, Δ is equal to 10 ms). If the frequency of the requests is high and so is the probability of collisions, this test will not often trigger $\mathbf{R_O}$ and thus will limit the effects of the collisions.
- The *Result* test triggers $\mathbf{R_O}$ except if there was at least one collision during the last two consensus. As for the *Time* test, the *Result* test will tend not to trigger $\mathbf{R_O}$ if the frequency of the requests is high.
- The *Random* test triggers $\mathbf{R_O}$ by using a fixed probability of triggering and so does not use any knowledge to decide whether to trigger $\mathbf{R_O}$ or not (in our simulations, the probability of triggering $\mathbf{R_O}$ is set to 0.8).

If the frequency of the requests is high and so is the probability of collisions, this test will not often trigger $\mathbf{R_O}$ and thus will limit the effects of the collisions. As for the *Time* test, the *Result* test will tend not to trigger $\mathbf{R_O}$ if the frequency of the requests is high.

3. **Unrealistic Criteria** The unrealistic tests are tests that can be computed on the log but can not be implemented as they rely on a knowledge of the future.

   - The *Optimal* test triggers $\mathbf{R_O}$ only if there is not collision in the future (as defined by the simulation algorithm) and so will give the best possible performance (in the case of the simulations).
   - Unlike the *Optimal* test, the *Worst* one triggers $\mathbf{R_O}$ only if there is not a collision (as defined by the simulation algorithm) and does not trigger it otherwise.

   We use these two tests as a scale for measuring the performances of the five other tests: If t is the mean duration of a consensus for an activation of $\mathbf{R_O}$ according a given criterion C, t_{worst}, the mean duration according the *Worst* criterion and $t_{optimal}$, the mean duration according the *Optimal* duration, the gain for C is set to $(t_{worst} - t)/(t_{worst} - t_{optimal})$.

6.7.2 Results and analysis

The behavior of the leader C_L leads to classify a consensus instance c in one of the there following classes.

1. *Consensus immediate*: C_L does not evaluate the triggering criterion because an initial value was already available at the end of the consensus $c-1$. $\mathbf{R_O}$ is not activated and the duration of consensus c is equal to D_{norm}.

2. *Consensus Type 1*: C_L evaluates the triggering criterion to false. $\mathbf{R_O}$ is not activated and the duration of consensus c is equal to D_{norm}. The prediction is either right or wrong. Of course, during the computation, C_L can not know (even a posteriori) if a mistake has been done or not. However, during our simulation we distinguish the predictions that are a success from those that are an error (a gain was possible).

3. *Consensus Type 2*: C_L evaluates the triggering criterion to true. $\mathbf{R_O}$ is activated. If no collision occurs, the activation is a success and the duration of consensus c is equal to D_{succ}. Otherwise the activation is an error and the duration of consensus c is equal to D_{fail}.

During our analysis, each consensus instance belongs to one of the five following classes: immediate, type 1 success, type 1 error, type 2 success, or type 2 error. To evaluate the risk of collision, we adopt the following rule. A collision occurs during the consensus instance c if $\mathbf{R_O}$ is activated during this consensus and at least another proposal is generated before the end of the consensus. Clearly, this choice leads to an overestimation of the risk.

Two other weaker definitions of the risk have been studied. In both, the risk of collision is no more the same during the whole execution of the consensus instance. In our second estimation of the risk, we consider that a risk exists only during the first half of the execution and is null after. In our third estimation of the risk, we consider that the level of risk decreases uniformly throughout the execution. Of course, the weaker the risk is, the higher the observed gain is. Yet, our results (relative placement of a criterion according to the optimal and worst cases) are slightly impacted by the choice of a definition rather than another. Therefore, the results presented here are those obtained with our first definition.

Table I.3 shows all the results for the different criteria, the different contexts and the two logs. We also measure the different percentages of successes and errors and the percentage of immediate consensus for each triggering criterion in each context and for the two logs. This gives us more details about the quality of the triggering tests. The table 6.6 resumes all the different percentages.

	Original log					Compacted log				
	Never	*Always*	*Random*	*Time*	*Result*	*Never*	*Always*	*Random*	*Time*	*Result*
Context C_{L5}	4.8%	95.1%	77%	89.7%	93.8%	25.2%	70.7%	60.8%	52.3%	69.7%
Context C_{L11}	19.9%	80%	67.9%	77.3%	80.1%	62.9 %	31.8%	37%	62.2%	35.1%
Context C_{W5}	18.9%	80.4%	68%	79%	78.8%	53 %	42.8%	44.7%	44.3%	44%
Context C_{W11}	29.9%	69.5%	61.4%	68.8%	70.1%	67.1 %	29.9%	36.9%	33.1%	33.5%

Table 6.5: Duration gains on the scale *Worst* (0%) - *Optimal* (100%)

	Original log							Compacted log						
Context C_{L5}	*Optimal*	*Worst*	*Always*	*Never*	*Random*	*Time*	*Result*	*Optimal*	*Worst*	*Always*	*Never*	*Random*	*Time*	*Result*
% immediate	2.9%	3.4%	3%	3.3%	3.1%	3%	3%	37.1%	43.5%	39.6%	41.6%	40.3%	41.2%	39.6%
% error 1	0%	94.2%	0%	94.3%	18.9%	5.8%	1.6%	0%	47.8%	0%	49%	10%	25.5%	2.6%
% success 1	2.4%	0%	0%	2.4%	0.5%	0.3%	0.2%	10%	0%	0%	9.4%	1.9%	7.2%	1%
% error 2	0%	2.4%	2.4%	0%	1.9%	2.1%	2.2%	0%	8.7%	9.3%	0%	7.4%	2.3%	8.3%
% success 2	94.7%	0%	94.6%	0%	75.6%	88.8%	93%	52.9%	0%	51.1%	0%	40.4%	23.8%	48.5%
Context C_{L11}	*Optimal*	*Worst*	*Always*	*Never*	*Random*	*Time*	*Result*	*Optimal*	*Worst*	*Always*	*Never*	*Random*	*Time*	*Result*
% immediate	3.3%	3.7%	3.4%	3.6%	3.5%	3.4%	3.4%	43.3%	49.6%	48.3%	45.5%	47.9%	45.9%	48.1%
% error 1	0%	93.7%	0%	93.7%	18.8%	5.6%	1.8%	0%	41.4%	0%	44.2%	8.6%	21.1%	2%
% success 1	2.7%	0%	0%	2.7%	0.5%	0.3%	0.2%	10.8%	0%	0%	10.3%	1.8%	7.5%	0.8%
% error 2	0%	2.6%	2.6%	0%	2.1%	2.4%	2.4%	0%	8.8%	9.2%	0%	7.5%	2.7%	8.5%
% success 2	94%	0%	93.9%	0%	75.1%	88.3%	92.2%	45.9%	0%	42.5%	0%	34.2%	22.9%	40.6%
Context C_{W5}	*Optimal*	*Worst*	*Always*	*Never*	*Random*	*Time*	*Result*	*Optimal*	*Worst*	*Always*	*Never*	*Random*	*Time*	*Result*
% immediate	12.9%	14.8%	13.4%	14.4%	13.6%	13.4%	13.5%	93.1%	93.7%	93.4%	93.4%	93.4%	93.4%	93.4%
% error 1	0%	77.1%	0%	77.4%	15.6%	1.9%	4%	0%	4.1%	0%	4.3%	0.9%	0.3%	0.4%
% success 1	8.4%	0%	0%	8.2%	1.6%	0.3%	1.2%	2.4%	0%	0%	2.3%	0.5%	0.2%	0.2%
% error 2	0%	8.1%	8.3%	0%	6.6%	8%	7.1%	0%	2.2%	2.3%	0%	1.8%	2.1%	2.1%
% success 2	78.7%	0%	78.3%	0%	62.6%	76.4%	74.2%	4.5%	0%	4.3%	0%	3.4%	4%	3.9%
Context C_{W11}	*Optimal*	*Worst*	*Always*	*Never*	*Random*	*Time*	*Result*	*Optimal*	*Worst*	*Always*	*Never*	*Random*	*Time*	*Result*
% immediate	13.6%	14.9%	14.1%	14.4%	14.2%	14.1%	14.2%	93.3%	93.7%	93.6%	93.5%	93.6%	93.6%	93.6%
% error 1	0%	76.4%	0%	76.8%	15.4%	1.8%	4.1%	0%	4%	0%	4.1%	0.8%	0.3%	0.4%
% success 1	8.9%	0%	0%	8.8%	1.7%	0.3%	1.3%	2.5%	0%	0%	2.4%	0.5%	0.2%	0.2%
% error 2	0%	8.7%	8.8%	0%	7%	8.5%	7.5%	0%	2.3%	2.3%	0%	1.9%	2.1%	2.1%
% success 2	77.5%	0%	77.1%	0%	61.7%	75.3%	72.9%	4.2%	0%	4.1%	0%	3.2%	3.8%	3.7%

Table 6.6: Percentages of error and success for the different contexts and triggering criteria.

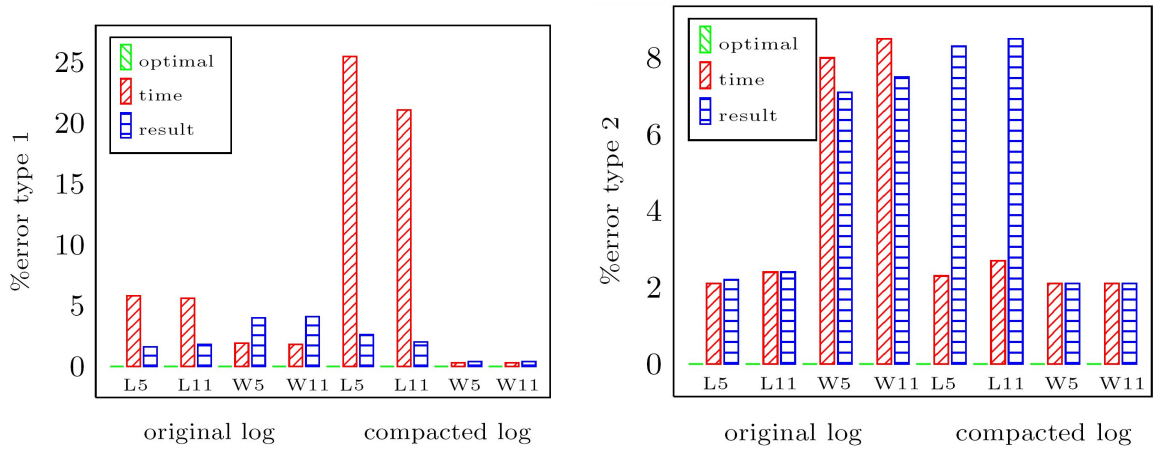
Table 6.6 gives a huge array of rates from which we extract the main information and we use it for a better representation in Figure 6.10 and Figure 6.11.

Regarding the original log, for all the contexts, the gains are high with *Random*, *Always*, *Time* and *Result*. The comparison of these gains with the gain obtained with the *Never* criterion unambiguously demonstrates the interest of $\mathbf{R_O}$. When considering the compacted log, the gain remains high for the C_{L5} context with the *Always*, *Random*, *Time* and *Result* criteria; it becomes less than the gain with the *Never* criterion for the three other contexts C_{L11}, C_{W5} and C_{W11}.

We also measure the rates of successes and errors (type 1 and 2) and the rate of immediate consensus for each triggering criterion in each context and for the two logs. Fig. 6.10 shows the rates of error type 1 and 2 with the *Optimal*, *Time* and *Result* criteria. Obviously, these rates are equals to 0 for the *Optimal* criterion (recall that *Optimal* is an unrealistic criterion that makes no error and only serves as a comparison point for the other criteria). Fig. 6.11 shows the rates of immediate consensus with the *Optimal*, *Time* and *Result* criteria.

Impact of the failure/success ratio on the gain.

In the case of the original log, the slight diminution of the gain from the C_{L5} context to the C_{L11} context can easily be explained by the failure/success ratio (see Table I.2) that is significantly higher for the C_{L11} context than it is for the C_{L5} context (for C_{L11}, a lot of successes in the activation of $\mathbf{R_O}$ are required to compensate for failures). Similarly, for the C_{W5} and C_{W11} contexts, the lower gain (observed for the original log) can also be partially explained by the increase of the ratio.



(a) (a) (b) (b)

Figure 6.10: Rates of error (type 1 and 2).

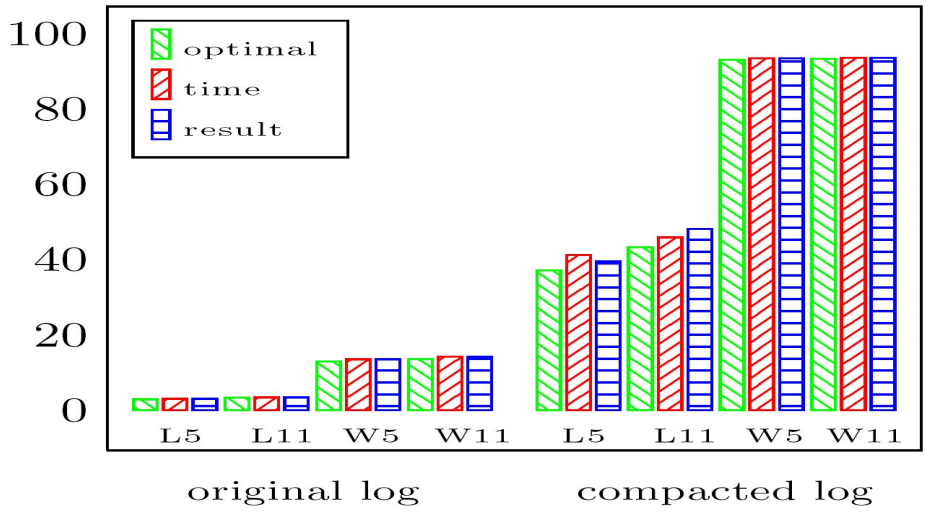


Figure 6.11: Rates of immediate consensus.

Impact of errors (type 1 and 2) on the gain.

We observe on the left part of Figure 6.10 (a) the increase of errors type 2 for the C_{W5} and C_{W11} contexts with the *Time* and *Result* criteria; this increase necessarily impacts the gain as the number of consensus with duration D_{fail} raises. As explained before, this is not true for C_{L11} for which the ratio augmentation is solely responsible of the diminution of the gain. Let's consider now the effects of the rate of errors type 2 and type 1 for the compacted log with the *Time* and *Result* criteria. With *Time* and for the C_{L5} and C_{L11} contexts, we observe a high rate of errors type 1 (above 20%) and a low rate of errors type 2 (under 3.5%). As seen before, an error type 1 leads to a consensus duration D_{norm} instead of D_{succ} (error type 1 means that $\mathbf{R_O}$ should have been triggered for this consensus and that it would have succeeded). Consequently, an error type 1 is penalizing when $D_{norm} >> D_{succ}$. For the C_{L5} context, D_{norm}

is 46% above D_{succ} while it is only 19% above for the C_{L11} context. For this main reason, the gain for C_{L11} is much better than the gain for C_{L5} despite the bad ratio failure/success exhibited by C_{L11}.

With the *Result* criterion and for the C_{L5} and C_{L11} contexts, we observe a low rate of error type 1 (under 2.5%) and a moderate rate of error type 2 (around 8%). Again, the augmentation of the ratio failure/success (from C_{L5} to C_{L11}) impacts the gain (69.7% for C_{L5} vs. 35.1 % for C_{L11}). It was already the case in the original log (rate of error type 2 around 2%). In the compacted log, this is even more obvious (rate of error type 2 around 8%).

Impact of immediate consensus on the gain.

In Figure 6.11, for the compacted log, we observe a high rate of immediate consensus (around 40%) for the C_{L5} and C_{L11} contexts and a very high rate (above 90%) for the C_{W5} and C_{W11} contexts. Consequently, for C_{W5} and C_{W11}, very few consensus will be eligible for the $\mathbf{R_O}$ optimization whatever is the retained triggering criteria. Here, the gains obtained for C_{W5} and C_{W11} are not significative because our way of computing the gain force it to spread between 0% (worst) and 100% (optimal) even if the values are closed to each other; indeed, for C_{W11}, the consensus mean duration (with *Optimal*) is just 2.6% below the consensus mean duration with *Worst* (as a matter of comparison, for the original log and for the C_{L5} context, the difference between *Optimal* and *Worst* reaches 44%). Whatever the used criterion, the adaptive behavior of the protocol is mainly due to the fact that immediate consensus are more frequent when the protocol is more solicited.

6.7.3 Which criterion?

So far, we did not evaluate the *Random* criterion. We introduced it in order to check if the results obtained with the *Time* and *Result* criteria were due more to their acuity rather than to the number of activation of $\mathbf{R_O}$ they induced. For the original log, the *Random* criterion is always below the *Time* and *Result* criteria. For the compacted log, *Random* is between *Result* and *Time* for the C_{L5} and C_{L11} contexts; as explained before, the gains for C_{W11} and C_{W5} are not significative. The *Time* criterion requires to tune the duration of the parameter Δ to fit the characteristics of the context. Obviously, 10 ms is a good choice for C_{L5} and C_{L11} but not for C_{W5} and C_{W11}. The *Result* criterion requires no tuning and allows to obtain a rather important gain whatever the context. A combination of criteria (conjunction or disjunction) has been also evaluated. However, this does not improve significantly the results and requires to select *a priori* if it is better to increase or to reduce the number of activation of $\mathbf{R_O}$.

Evaluation of the collision risk

The results obtained in our simulations depend on the way we evaluate whether two or more requests generate a collision when the risky optimization is triggered. For the results presented, we estimate that a collision occurs when two consecutive requests arrive in less than the average consensus time for the context (when $\mathbf{R_O}$ is triggered). It is clearly an overestimation of the collision risk. We try two other collision risk parameters and we launch the set of simulations for both theses parameters. First, we reduce the value of the time parameter by fixing it at half the value of the average consensus time. Secondly, we change the law:

we consider that a collision occurs with a probability inversely proportional to the difference between the arrival time of two requests if the difference is less than the average consensus time and zero otherwise. In this case, when the optimization is triggered, we consider all the requests that have a positive probability to generate a collision for the consensus and generate a random number between 0 and 1 for each. If for one of the requests, the random number is less than the calculated probability, we consider that a collision occurs for the consensus.

The two other collision risk parameters we use clearly reduce the number of collisions in our simulations. The results obtained for these two parameters are very close. The gain obtained by the *Always*, *Result* and *Time* triggering criteria are closer to the *Optimal* depending on the context and the log, the gain increases by 0.5% to 6% for these criteria. The success and error percentages are better too. These two other collision risk parameters do not change the relative results of the triggering criteria for the different contexts and the different logs and the analysis is the the same for these collision risk parameters.

6.8 Final remarks

This chapter presented the evaluation of the Paxos-MIC protocol, carried out in different testing scenarios that are relevant for the protocol's behavior. The main purpose of the experimental phase was to determine the factors that influence the protocol's performance when optimization $\mathbf{S_O}$ or both optimizations $\mathbf{S_O}$ and $\mathbf{R_O}$ are activated. The second part of this chapter was dedicated to the study of optimization $\mathbf{R_O}$ and more precisely, the triggering criteria and the possibility to predict collisions. We considered an application for securing a Web server and showed that simple triggering criteria do not allow to predict accurately collisions but they are precise enough to adapt dynamically the behavior of the protocol to the current context. All our results demonstrate that there is a real interest in using $\mathbf{R_O}$.

Part IV

Sequence of Decisions - Support for Transactional Mobile Agents

Chapter 7

Sequences of Decisions as a Support for Transactional Mobile Agents

Contents

MANY distributed applications that require fault-tolerance are built upon a sequence of decisions generated by repeatedly invoking a consensus building-block. In this chapter, we study the context of transactional mobile agents and propose a solution to support the execution of transactions in ad-hoc networks. We begin by describing the general context of transactions and the atomic commit problem. Further, we outline the underlying principles of the mobile agent technology. In such a context, we identify two services that support the execution of transactional mobile agents and show how these services can be implemented by relying on an everlasting sequence of decisions. The Paxos-MIC protocol is in charge of building such a sequence.

7.1 Context

A transaction is a computation unit that combines multiple instructions into a single atomic action. Transaction processing is increasingly used in designing distributed applications. In a distributed system, a transaction can be defined as a composition of several separate requests, each request being directed to a particular remote node. As the whole execution of a transaction may modify data across multiple nodes and organizational boundaries, transactions are well suited to develop e-business applications that involve autonomous and heterogeneous participants into a collaborative process. In particular, the atomicity property ("All or nothing") is a key for solving booking problems when the required resources are distributed all over a wide geographical area.

When a node executes a request, the outcome depends on its local state but it may also depend on the inputs provided by its surrounding environment. In particular, sensors are now embedded in many devices: they provide multimedia data (microphone, camera) and sense their environment (ambient light, temperature, airflow, noise, motion, positioning). Moreover, systems are more and more pervasive in our everyday's lives. Wireless mobile devices (such as a smartphone, a laptop, or a tablet) allow to access the Internet from anywhere and also to participate in dynamic ad-hoc networks consisting of other devices located in the neighborhood. As a consequence, the specification of a request can now also refer to these surrounding (mobile and static) sources of information. A transaction composed of n requests may involve n main nodes but also several sensors and people that are either located or in transit in one of the n identified geographical sub-areas. Furthermore, as a transaction is described by a sequence of requests its execution determines an itinerary that allows to reach successively the n separated sub-areas [58]. For example, in the context of a mall, a sub-area may correspond to a shop. A transaction can be used to organize the future walk of a customer: executing the transaction leads to identify n shops that can satisfy the client's requirements and make the necessary bookings of items. To illustrate this point, let us consider the following booking scenario.

> *When his car arrives at a mall, a client may launch a transaction denoted P;R;M;T which expresses his following booking requirements. Just before lunchtime, this client wants to find a shady place in the parking lot of the shopping center (request P, for Parking). Then, he wants to have breakfast on the terrace of a crowded but quiet restaurant, if a table protected from wind and sun is available (request R, for Restaurant). Today is the birthday of his girlfriend Christina. After lunch, he wishes to purchase an album from any metal band among the top ten best-selling albums of the week but only if this album appears on the playlist of at least one customer who has about the same age as Christina (request M, for Music). Furthermore, he wants to offer her a black T-shirt on which the band name is printed (request T, for T-shirt).*

The transaction $P; R; M; T$ is composed of four requests. For each of them, at least one node able to satisfy the request has to be discovered. In our example, such a node is acting on behalf of a shop representative. Sensing devices are used to attest that all the specified requirements are met (for example, the lack of noise in a restaurant). People in the vicinity are also consulted (for example, their payload lists are accessed). In our approach, we propose and evaluate some strategies that will enable this scenario to become a reality in the very near future.

The sources of information that are consulted during the execution of a transaction are numerous and unpredictable. As a consequence, a centralized solution where all the produced data are gathered by a few dedicated servers has to be avoided. A centralized storage makes sense only if the data is very specific and relevant to a wide range of people. For example, an application such as the monitoring of traffic, weather and road conditions may benefit from a centralization of informations produced within a large geographical area. In our transactional scenario, the nature of the data is rather varied and they are unevenly produced by unregistered devices. The relevance of each information is not recognized by all the users: the noise in a restaurant or the shadow in a parking lot are time-fluctuating information that are critical for only a limited number of users. Continuously gathering a large volume of data is clearly counterproductive when each data concerns neither a geographical hotspot nor a thematic hotspot. Moreover, in general, in the case of sensors, only the last available value is of interest (not the whole sequence of values produced in the past). To choose between a centralized and a decentralized storage of the data, security concerns have also to be taken into account. In our scenario, the lack of formal relationships among the different traders is a major obstacle in the way of developing a centralized solution. Indeed, some shops may have competing business activities and thus may be reluctant to entrust to a third party the management of business information, especially if this third party is also related to competitors. For all the above reasons, data have not to be transferred to a central entity but they should remain stored on the sites where they have been produced.

The agent technology aims at moving the code where the data are produced [50]. Consequently, this technology seems appropriate. A mobile agent is a computer program that can migrate autonomously from node to node, in order to perform some computation on behalf of the codes's owner. In our context, the code's owner is the customer who issues a transaction and expects a firm commitment of n traders about its n requests. The agent is in charge of exploring a geographical area (*i.e.* the mall). In order to move within this area, it relies on both fixed and ad-hoc networks. Indeed, an agent is a proven solution to cope with the dynamic changes that continually modify the topology (connections and disconnections, node's movings). For example, in such contexts, this technology was adopted to solve routing [6, 35] and service discovery [41, 22]. Herein, for each of the n stages of the transaction, the agent must identify a node (called *a place*) able to satisfy the corresponding request. A node is defined as either a static or a mobile device owned by a trader. By definition, a node is a computing unit that provides an appropriate infrastructure to support a mobile agent migrating to and from that location. For a given request, a node is able, first, to test if it may satisfy the request and, second, to execute the corresponding work if needed. During both the test and the request's execution, a node may interact with its surrounding environment and in particular with available devices located nearby. As sensor-generated data fluctuate both with time and with the location of a device, the values taken into account are those available at the time of the visit of the agent. Regarding its communication capabilities, during the visit of an agent, a node can provide an IP address which can be used later to contact this node directly. The concept of transactional mobile agent [51] has been introduced as a mix between the agent technology and the transactional model. During its move, the agent discovers and visits n places that satisfy the n requests. During the agent's visit, a node records enough information so that it can subsequently either commit or abort the transaction.

Different works on transactional mobile agents [28, 29, 52, 49] have identified two

problems that can not be solved by the agent alone. The first one is related to the reliability of the agent. Agents and nodes may fail. More precisely, we consider that they may fail by crashing. Failure detection and recovery procedures have to be defined. A classic solution consists in using the last visited node to monitor the state of an agent once it moves to another place. In that case, the last visited node which is hosting a failure detection mechanism becomes also a critical point of failure and thus, it has also to be monitored. One thing leading to another, all the nodes already visited by the agent (and the client who initiates the transaction) have to form a chain where each node observes its successor and may act to ensure the existence of at least one alive agent. When the agent is suspected to be crashed or when the monitoring chain is broken at a point, a new agent is created by the node that suspects the occurrence of a failure. Note that in an asynchronous distributed system where failure detectors are sometimes unreliable, the generation of a new agent in case of problems no more guarantees its uniqueness. Since the client is the first node in the monitoring chain, it has to remain connected during the whole execution of the transaction. The second problem for which the agent needs assistance is related to the atomic validation of the transaction. Once an agent has identified and visited n nodes able to satisfy the n requests of a transaction, it has to delegate the supervision of the validation process to an external reliable entity. For several reasons, this final task should not be made solely by the agent. First, by definition, an agent is a moving entity that executes a local code on the node where it is located. But a validation protocol is typically a distributed service that requires communications between its initiator and the n involved nodes. Second, the computing power of an agent depends on its hosting node which may have limited processing, storage and power resources. Third, a validation protocol relies on the fact that a single coordinator exists or, more generally, on the fact that any coordinator acts in a manner consistent with what has been done by previous coordinators. As mentioned before, to tackle dependability issues, multiple agents are sometimes created. As they do not know each other and sometimes follow different itineraries, ensuring consistency between them is impossible. Last but not least, the outcome of a transaction (commit or abort) has to be kept in stable storage. Ensuring the persistency of data is not a task that can be assigned to a mobile agent. To address all the above remarks, a centralized and reliable entity can be defined to assist all the agents in their monitoring, validation and logging tasks.

The duration of an execution depends on n, the number of requests specified within the transaction. Indeed, the greater the value of n, the longer the path followed by the agent to find n places. In commercial applications, the execution of a request typically results in the booking of some resources [61]. As a consequence, reservations made during the visit of an agent can be canceled very late (when the transaction is aborted). In general, a long-lasting booking of resources is not desirable. It can be detrimental to the trader's interest. Indeed, while a resource is blocked, the seller may refuse to satisfy other requests that require the same resource. If the transaction (for which the trader was prepared to satisfy a request) aborts, he may miss many sales. Consequently, to be widely accepted, a solution must ensure that the trader keeps control over its resources for as long as possible.

7.2 Transactions, Requests, and Nodes

Although the transactional model is widely known and adopted in many application domains, this section aims at briefly recalling some related concepts and definitions. Some notations and the terminology used within this chapter are also introduced.

7.2.1 Distributed Transactions

The notation T_α refers to a transaction uniquely identified by α. We assume that a client generates a different identifier for each new transaction it creates and starts. To ensure the uniqueness of a transaction's identifier within the whole distributed system, a client can include its own identity as part of each identifier it produces. A transaction is defined as a sequence of requests. Let us assume that the transaction T_α consists of n successive requests: $T_\alpha \equiv R_\alpha^1; R_\alpha^2; \ldots R_\alpha^x; \ldots R_\alpha^n$. The integer value x is called the stage of the request R_α^x while n represents the length of the transaction. Going back to the example described in the Introduction, the transaction $P;R;M;T$ is initiated by a client before its visit to a mall. It consists of four requests denoted respectively P (for Parking), R (for Restaurant), M (for Music) and T (for T-shirt). The execution of a transaction requires finding, for each request, a node that can satisfy it. The discovery process is detailed in Section 7.2.2.1.

The transactional model is characterized by the four ACID properties (*atomicity*, *consistency*, *isolation*, and *durability*). When these properties are satisfied, a transaction can be considered as a single logical operation whose execution is reliable despite the possible hardware and software failures. A transaction is *atomic*: the rule *"all or nothing"* ensures that either the entire transaction is executed or none of the requests are executed. A transaction is *consistent*: either it has no impact on the distributed system or its execution takes it from one consistent state to another. In particular, as a request can be computed by different alternative nodes, each request has to be executed by just one of them. In its most restrictive definition, the *isolation* property states that a transaction should not observe changes made by another transaction if the latter is not yet complete. Yet, for the purpose of efficiency, weaker definitions are usually adopted []. Finally, the *durability* property guarantees that the effects of a committed transaction are never lost even if failures occur in the future.

The lifetime of a transaction can be divided into two phases, called the *execution* phase and the *commitment* phase. During the first phase, for each request, a node able to execute it has to be found. The second phase begins when n discovered nodes have processed the n requests: an atomic commitment protocol is used in this final phase to ensure transaction atomicity.

7.2.2 The Execution Phase (1^{st} phase)

7.2.2.1 Discovery of alternative nodes

A request R_α^x expresses the client's needs with regards to products and services that may be provided by a single node. However a request does not identify the node that will be responsible for meeting them. A request is a purely local transaction: its execution corresponds to a sequence of read and write statements that are applied atomically to a local information system. In our context, a node is a computing unit that acts as a representative for either

an economic organization (for example, an enterprise or a shop) or a geographical area (for example, an entrance of a mall).

In the example, the transaction involves four nodes that act respectively as a representative for the parking lot, a restaurant, a music store and a clothing store. To hide the complexity of each local environment, we assume that the local execution of each request is under the responsibility of a single node. Yet this node can coordinate read operations performed on several other computers, mobile devices and sensors that belong either to the same organization or to external people in transit in this area. For example, in a restaurant, sensors are disseminated and queried to determine the temperature or the noise level. In the music store, the tastes of other clients are taken into account by accessing their personal playlists.

As the identities of the n nodes are not specified in the transaction itself, a mechanism for discovering sites that are likely to participate in the transaction is required. In order to be selected to execute a request, a node has usually to be registered in public repositories as a possible *alternative* for this type of request. In a commercial context, such a registration is natural and is part of the promotion process and business development: it helps to inform potential customers about the services and products offered. The discovery process may rely on various mechanisms [45]. The fact that a node is considered as an alternative node does not imply that it can currently satisfy the client's needs. For example, a registered restaurant is perhaps already full or its noise level is too high compared to the wishes of the client.

For a given request, once different alternative nodes have been discovered, their ability to execute the request is evaluated in an order that is usually not left to chance. For example, minimizing the physical distance between any two consecutive nodes involved in the transaction can be an extra wish of the client [58]. To take into account such an additional requirement, a greedy strategy is often chosen. The different stages of the request are analyzed in sequence: looking for a solution at stage $x+1$ begins only once a node N_i that satisfies the request R_α^x has been found. The alternative nodes that may execute the request R_α^{x+1} are considered one after another from the nearest to farthest from N_i till one of them is able to satisfy the request. This greedy strategy often, but not always, yields an optimal solution where the physical path that connects all the selected nodes is minimal.

For each stage x, the number of alternative nodes that may satisfy a request R_α^x depends on the demand of the client. Obviously, a client that is looking for a place to eat is less difficult to please than a client looking only for a pizzeria. If the client's request is too precise and too restrictive, his request may not be satisfied and, as a consequence, the whole transaction is unfulfilled. But if the request is too vague and too permissive, it may be satisfied in a way that is not ideal for the client. To solve this dilemma, levels of satisfaction can be defined for each stage. In that case, a request R_α^x is now represented by k_x alternative requests that are denoted $R_\alpha^{x_1}$, $R_\alpha^{x_2}$ $\ldots$ $R_\alpha^{x_{k_x}}$. The request $R_\alpha^{x_{k_x}}$ is the one that maximizes the client's satisfaction while the request $R_\alpha^{x_1}$ is the one that meets his minimal requirements. At each stage x, the requests are examined in descending order from k_x to 1 till a node that satisfies a request $R_\alpha^{x_k}$ with $k_x \geq k \geq 1$ is found. A request $R_\alpha^{x_k}$ is less demanding or at least different from the $k_x - k$ requests previously examined. Thus by defining several levels of satisfaction for each stage, the chances of finding a node that is able to meet one of these alternative requests increases while the discovery process is still based an order of preference expressed by the client himself. In our example, regarding the request R of stage 2, the client may wish to eat in a restaurant with a menu under 15 euros. Yet if it is impossible to

satisfy this first request, he may agree to relax its budget constraint and to accept any menu under 25 euros (*i.e.* an alternative request with a lower satisfaction level). In the context of an atomic transaction, this feature is important as it may avoid to abort a long lasting transaction (*i.e.*, a transaction with many stages) just because a single request (for which the client is ready to make concessions) can not be satisfied. Clearly, the concept of satisfaction level is of practical value. Yet, as it just impacts the discovery process, we often refer only to a single request R_α^x per stage (unless otherwise noted).

7.2.2.2 Execution of a request by a node

When an alternative node N_i is contacted to execute a request R_α^x, it has to assess whether it can satisfy the request and run it once. If the node can not satisfy the request, the outcome is negative and the node keeps no information about its failed attempt to execute the request. Otherwise, if enough resources are available and all the operations are successful, the outcome is positive. In that case, the required local resources are booked on a temporary basis and the result of the execution, as well as how this computation may later affect the local state of the node are logged in a versatile memory. At this point, the node has made no firm commitment. A booking can be canceled at any time and for any reason (timeout expired, reallocation of the resources to another request, ...).

When the outcome is positive, the result returned by a node contains information that may be used by dependent requests. By definition, a request R_α^y depends on a request R_α^x (with $x < y$) if R_α^y refers to some variables that are initialized during the execution of R_α^x. For example, in the transaction P; R; M; T, the booking of a table in a restaurant (request R) requires to know the timeframe during which a parking space is assigned (request P). Similarly, to buy a T-shirt with the name of the band printed on it (request T), the selected disc of music must be known (request M). To cope with these frequent dependencies, an additional rule has to be respected. Requests of a same transaction have to be executed sequentially in accordance with their stage number and not in parallel. This constraint can be relaxed [57] only when the lack of dependency is proven. Herein, we assume that no exception to the rule is allowed: the requests are examined sequentially. Thus, when a node N_i is contacted to execute a request R_α^x, the parameters provided to it mention the results obtained during the execution of all requests of lower stage (*i.e.* requests R_α^y such that $1 \leq y < x$). This rule has a negative impact on the duration of a transaction which depends mainly on the length n of the transaction. As a consequence, for each request R_α^x, the period during which resources are booked depends also the stage number x: the lower this number is the longer the booking duration should be. The execution phase ends when the execution of the last request R_α^n ends successfully or when no node can be found for a given level.

7.2.3 The commitment phase (2^{nd} phase)

An atomic commitment protocol is launched once n participating nodes have been identified. Such a protocol aims at ensuring agreement among these n nodes on whether to commit or abort (roll back) the transaction. Many atomic commitment protocols rely on the definition of a (static or dynamic) leader in charge of coordinating the n nodes. During a voting phase, a coordinator attempts to prepare all nodes to take the necessary steps for either committing or aborting the transaction. More precisely, each node votes either *Yes* if it

agrees to commit the transaction or *No* if it appears that the transaction has to be aborted. The reply *No* is always returned by a node if it has detected a local problem that may prevent the execution of the assigned request to end properly. During a commit phase, the coordinator decides whether to commit or to abort the transaction. Then it notifies this outcome to the n nodes. To decide commit, the coordinator must received n votes *Yes*. Otherwise, if the coordinator receives at least one vote *No* or if some votes are missing (due to the crash of a node, its slow execution speed, the lost of a message, or high transfer delays) deciding to abort is mandatory. Whatever the decision, the n nodes must comply and perform the necessary actions.

If a node replies *Yes*, he agrees to fully and properly execute its assigned request whatever the circumstances. In particular, if it answers *Yes*, the corresponding bookings become locally irrevocable: only an abort of the transaction may result in revoking them. Since a node that returns *Yes* relinquishes control over its allocated resources, it must be informed of the final decision (Commit or Abort) whatever the circumstances. A commitment protocol generally ensures a stronger property: every node contacted (by a coordinator) during the commitment phase is informed of the final decision. The coordinator knows the list of n nodes that have been contacted. Thus it can notify the outcome to all these nodes (*i.e.* even to those from which it has not received an answer). When the transaction aborts, no resource remains locked.

The atomic commitment protocol is executed by one or several processes called the transaction manager(s). Once n nodes have been identified, an interaction between a transaction manager and these n nodes occurs through one of the two following primitives: *Vote()* and *Outcome()*. Both primitives are called by the transaction manager. In both cases, a message is broadcast to the n nodes. Any message sent by a transaction manager to a node identifies (without any ambiguity) a visit previously made by an agent to this node: the message contains the transaction id α, the request id x and a tag that identifies an instance of a visit related to R_α^x. The primitive *Vote()* triggers the vote. In response, each node returns either the value *Yes* or the value *No*. The primitive *Outcome()* informs the nodes of the decision (*Abort* or *Commit*). In response, each node returns an acknowledgment.

In the proposed solution, an agent (and its possible replicas) may identify different sets of n nodes that satisfy the requirements expressed in the transaction. Rather than just checking the first one, the commitment phase can be revisited to increase the chance of reaching a positive outcome (*Commit*) by increasing the number of trials. For a given transaction, the atomic commitment protocol can be run multiple times till a significant decision (*Abort* or *Commit*) is adopted. During each execution a different set of nodes is analyzed. Of course, each execution is still carried out according to the general principles described in the above paragraph. Yet the outcome of an execution can now be one of the three following values: *Abort*, *Commit* or *Release*. For a given transaction, the last execution will reach a significant decision (*Abort* or *Commit*), while all the previous ones necessarily end with the outcome *Release*. *Abort* is decided only when it appears that no new set of n nodes is going to be identified.

7.2.4 The six possible states of a visited node

In the following, we focus on the behavior of an alternative node N_i involved in the execution of a request R_α^x. We will see later that the node N_i can be solicited several times for the same

request R_α^x. Each solicitation is called *a visit*. A visit is identified by a triplet $< \alpha , x , t >$: the two first fields identify the request while the last one, called a tag, allows to differentiate between visits.

In a trading context, a request often refers to local resources that are for sale (articles, deliveries, ...). If they are not available, the request can not be satisfied by the node. If the transaction commits, they are purchased and consumed by the client. As mentioned before, the resource allocation system has to be as flexible as possible. In particular, firm booking have to be done as late as possible. In our example, a shopkeeper can set aside an item (a parking space, a table, a disk, or a T-shirt). During a limited period of time, this product will be considered already sold or assigned. Yet, the node can reverse this temporary commitment.

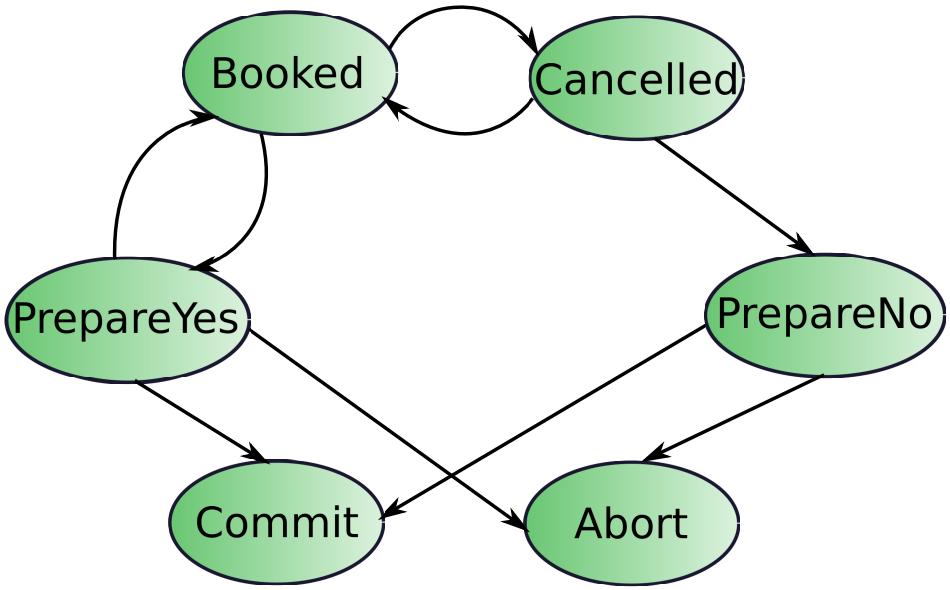


Figure 7.1: The possible states of a visited node and the transitions between them.

To keep control over its revocable and firm reservations, each node N_i manages a local table called *Tab_Visit* in the proposed solution. When a node is visited, it first assesses whether it can satisfy the request and run it once. Let us consider that N_i is able to satisfy the request R_α^x. In that case, N_i creates a new entry in the table *Tab_Visit*. An entry of the table contains three main fields corresponding to the identifier of the visit, the results generated during the successful execution of the request and the *state* which is initialized with the value *booked*. The state is the only flied which may change later either during the execution phase or the commitment phase. We define six different states that can be reached by the node N_i during the execution of the request R_α^x. The six states are called the *Booked* state, the *Cancelled* state, the *PrepareYes* state, the *PrepareNo* state, the *Commit* state, and the *Abort* state. The possible transactions between those states are depicted in Figure 7.1. While N_i remains in the *booked* state, the required resources are booked on a temporary basis and the result of the execution of R_α^x, as well as how this computation will affect the local state of the node are logged in a versatile memory. In the *Booked* state, a booking can be canceled at any time and for any reason (timeout expired, reallocation of the resources to another request, ...). The decision to cancel is the sole responsibility of the node itself. If the booking is invalidated,

the node N_i transits into the state *Cancelled*. But a new reservation (which is the same as the first) is still possible later. The duration of the stay in the state *Booked* is usually long and depends on the stage x of the request: the smaller the value of x the longer the duration of the stay. Therefore there is a real interest to make no firm commitment during this period. When N_i receives a *Vote* message related to this particular visit, it transits either to the *PrepareYes* state or the *PrepareNo* state depending on its reply (*Yes* or *No*). When the node receives an *Outcome* message related to this visit, it transits to the *Commit* state (respectively the *Abort* state) if the received message contains *Commit* (respectively *Abort*). Otherwise, when the *Outcome* message carries the value *Release*, the node N_i will either stay in the *PrepareNo* state or transit from the *PrepareYes* state to the *Booked* state. In this last case, the node comes back to its previous state and switches from a firm reservation to a cancelable one.

Note that the progress into one of the four last states is driven by the atomic commitment protocol. The node N_i must received a message from an entity involved in the atomic commitment protocol to transit into one of these states. The duration of the stay in either the *PrepareYes* or the *PrepareNo* state is the same for all the nodes involved in the transaction. Once arrived in either the *PrepareYes* or the *Commit* state, a node has to ensure the promised provision of resources.

7.3 Use of Mobile Agents

The *agent* technology aims at moving the code where the data are produced and stored. Therefore, such a technology is well suited to design decentralized solutions where the data remains on the owner's sites. Agents have been used in various contexts (LAN, sensor networks, ad-hoc wireless networks). In the context of non-persistent networks, agent based solutions have been proposed to solve *Discovery* and *Routing* problems [41, 22, 6, 35]. Thus the heterogeneity of the environment that characterized our targeted applications is not a major constraint and do not prevent the use of agents. Through the concept of *transactional agent*, It has been proved that transaction processing can also be implemented following the agent based approach: the concept of transactional agent has been studied in several works [51, 49]. When a client launches a transaction T_α, a mobile agent A_α is created. This agent is able to migrate autonomously throughout a defined geographical area. Within this region any node that offers services must also provide an execution environment that allows the agent to reach the node and to execute its code. Thus codes related to the n different stages of the transaction can be transferred and executed on remote sites. According to the usual terminology, a visited node is called a place as it corresponds to a context in which the agent executes. The node where the agent A_α is initially created corresponds to the place p_0. Starting from this initial place, the agent will move to discover a path (called an itinerary) that goes through n other places (denoted p_α^1, p_α^2, ..., p_α^n). An itinerary must satisfy the following property: if a place p_α^k is listed in the itinerary, the associated node has confirmed that it was able to execute the request R_α^k at the time of the visit of the agent (*i..e.* an entry has been created in the table *Tab_Visit* of the node).

Once the agent knows a complete itinerary (*i.e.* after the visit of the last place p_α^n), the agent stops moving and an atomic validation protocol can be launched. This strategy is called *Commit-At-Destination*: the validation of the transaction starts only when a set of n participants has been identified by the agent. In that case, a single decision is made for the

entire transaction. Another approach called *Commit-After-Stage* requires to decide at each stage of the construction of the itinerary. In that case, the commitment of a node is made during the visit of the agent and is irrevocable. This second approach leads to confirm bookings well in advance, especially in the case of the first visited places. For this reason we consider herein only the first approach as it allows to postpone the definitive booking of the local resources for as long as possible. In favorable circumstances, an agent that starts from p_0 migrates only n times to discover a complete itinerary. Its behavior can be summarized as follows. In each place p_α^k (with $0 \leq k \leq n-1$), the agent identifies an alternative node for the next request R_α^{k+1}. Then it migrates to this node which becomes the place p_α^{k+1}. Finally, before iterating to the next stage, it checks whether the new visited node is able to execute the request R_α^{k+1} and to book (for a minimum period) the required resources. In the above description, nothing complicates and delays the progress of the agent. Yet as mentioned before, two phenomena disrupt the creation of an itinerary. First, the node selected to be the next place is not always able to satisfy the request's requirements. Second, crash failures may affect the agent and the node currently visited. To cope with crash failure, monitoring mechanisms have to be created and activated by the agent itself all along its itinerary. More precisely, before leaving the place p_k to migrate to p_{k+1}, the agent creates a fixed agent called a *watch agent* which observes the agent during its attempt to move. A watch agent is in charge of executing a monitoring code on behalf of its creator once this one has leaved the place. In favorable circumstances, the agent will create a path of length n and will leave behind n watch agents (one per site visited). Failures are detected using classical detection mechanisms: to prove that it is still alive an agent sends periodically messages to the last created watch agent. When a watch agent receives no messages, it suspects that a crash has occurred on the next place. Of course, as the monitoring of the agent is performed by a single watch agent which may fail, this watch agent has also to be monitored. To solve this problem, all the watch agents form a chain of control that maps the last itinerary discovered by the agent. The watch agent of the visited place p_k is monitored by the watch agent of the previous place p_{k-1}. Of course, such a solution relies on the assumption that the first element in the chain (*i.e.* the place p_0) is either safe or under the control of another external entity that is reliable. Once this germ of reliability is established, the simple monitoring mechanism allows to detect failures (crash of a visited node, crash of the agent) and to create a new copy of the agent. As explained in section 7.1 ensuring *the availability* of a first place (also called *a source*, herein) is one of the two services that cannot be ensured by an agent alone.

The *atomic validation* is the second service that deserves to be under the control of some entities that are neither the visited nodes nor the mobile agent. Of course, the agent has to interact with these entities in charge of executing the atomic commitment protocol. Three types of messages are defined and called respectively *End*, *Stop* and *Double*. These messages are sent by the agent to the external entities. When the agent arrives at a destination (*i.e.* once an itinerary of length n has been found), it stops its execution and provide, within a message called *End*, the list of n visited nodes. A visited node is not always able to satisfy a request (lack of resources, requirements not met). After the visit of a place p_k that is not a destination ($k < n$), the agent visits different alternative nodes one after the other until it discovers a place that can satisfy the next request R_α^x. If no place is found, the agent stops the migration and sends a message *Stop* to indicate the prefix of an itinerary that can not be completed. Finally, for the same request, R_α^x, a node can be visited several times by different copies of

the agent. The failure detection mechanism is at the origin of these multiple visits. First of all, the failure detection mechanism is unreliable (as it is implemented in an synchronous environment). A correct agent can be erroneously suspected to be crashed by a watch agent located on the previous place. In that case, a new agent is generated and moves towards an alternative node while the former continues its way in the network. Both agents follow different paths but can later visit the same node regarding the same request. In that case, an agent is older than the other one. Thus, any node visited by the two agents can easily determine (at the time of the second visit) in which order it has received the two agents (the oldest and then the newest or the newest and then the oldest). A node authorizes an agent to progress only if its visit is the first one related to this request or if all the previous visits have been done by older agents. When a agent is not authorized to continue, it stop its execution and send a *Double* message to the external entities. This message contains the prefix of an itinerary that can not be completed. Note that a node keep track of all the visits already made (table *Tab_Visit*) even if only the most recent one is used to test if a new visit is allowed or refused. A tag is used to date the visits. For a visit corresponding to stage x of the transaction, the tag is a sequence of x integers $t_1\ t_2\ \ldots\ t_x$ where each t_y with $1 \leq y \leq x$ is the number of agents that have leaved the place p_{y-1}: this includes the first agent that has visited the place p_{y-1} and the possible $t_y - 1$ replicas generated by the watch agent located in the place p_{y-1}. In Figure 7.2, we illustrate a possible scenario in the case of the transaction $P;R;M;T$. False suspicions are depicted by dotted crossed lines while real crash are represented by continuous crossed lines. The *Stop* message contains a tag equal to 11. The *End* message includes a full itinerary and a tag equal to 2211. The *Double* message contains a partial itinerary and a tag equal to 211. Indeed we make the assumption that the second generated agent was slower than the third one. In the next section, we will show that the atomic commitment protocol take advantages of the full itineraries but also of the partial ones.

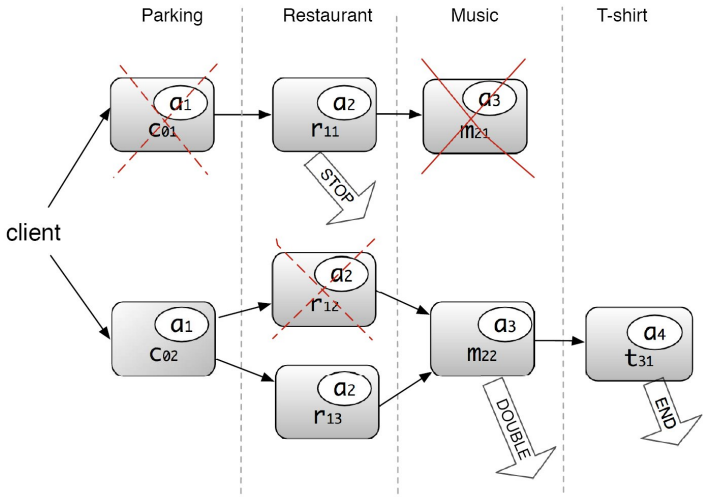


Figure 7.2: Multiple itineraries and messages.

The failure detection mechanism aims at detecting the failure of an agent. But more generally it detects any break in the monitoring chain of watch agents. When the problem is due to the failure of a watch agent, a new agent is created even if the previous one is still

alive (and potentially far away from the crashed node). Indeed this creation is essential as an new itinerary can only be build by a moving agent. The same failure detection mechanism is also used to cope with booking failure. When a place p_{k+1} decides to cancelled the booking related to a given visit (*i.e.* the most recent one), it sends a failure message to the watch agent located on the previous place p_k. This one behaves as if it was a crashed failure: a new agent is generated to create a new path.

7.4 Use of a centralized support

In the proposed solution, two services are provided by the cloud: *Availability of the Sources* (denoted AS) and *Atomic Commit* (denoted AC).

Whenever it wants to start a new transaction T_a, a client activates the **AS service** by sending a *Req_AS* message. Through this service, the client delegates to the cloud all the tasks related to the agent activation and the monitoring of the source places. As the progress of the transaction and the persistence of the final result (*Commit* or *Abort*) will be ensured by the two cooperating networks, the client does not have to remain connected while waiting for the outcome. The *Req_AS* message includes the identity of the transaction T_a (denoted *Tid*) and a descriptor (denoted *Desc*) where some static information about the transaction itself is stored: the number of stages (variable *Stages*), the code used to create an instance of the agent A_a in a source place (variable *Code*), information to identify some potential sources (variable $Places_0$) and a timeout whose use is detailed later (variable Δ). Periodically, the AS service will check the status (available or failed) of the source place(s) where it has previously activated an instance of the corresponding agent A_a. If none of these places is still available (or if no activation of A_a has occurred before), AS looks for another possible source place: it calls the function *NewPlace* with the variable $Places_0$ as a parameter. Such a call returns either a next source place or a special value $\perp$. We will not detail the mechanisms that may be used at this level. Note that, in a very simple solution, the variable $Places_0$ is a list of predefined places and the function *NewPlace* just returns one of the places not yet explored, if any, or $\perp$ otherwise. More sophisticated solutions based on dynamic discovery services can also be defined. Whatever the adopted solution, if this research is successful, the agent A_a is activated in the new discovered source place. If no alternative exists (*i.e.*, the returned value is $\perp$), a *Req_AC* message with an empty itinerary and a *State* field equal to *Stop* is directly addressed by the AS service to the AC service. In this way, the AS service informs the AC service that, due to multiple failures of source places, it cannot ensure that an instance of the agent A_a is able to migrate within the ad-hoc network to reach a non faulty place from which it will send a *Req_AC* message (with a *State* field equal either to *End* or *Stop*). In our solution, the AS service must keep the identities of all the source places where an instance of the agent A_a was already created. A failure detector periodically observes the status (available or failed) of each of them and maintains a suspicion list. As the failure detector is not necessarily reliable, all the source places already created (and not only the last one) are monitored. Of course, if the failure detector is perfect (no erroneous suspicion), just one source place, namely the last place where the agent A_a was activated, has to be checked. To mask these details we consider a boolean function *Alive(Kp)* that returns false if and only if all the source places contained in the list of places *Kp* are suspected to be failed. The AS services implemented at the cloud level manages only the source places (*i.e.*,

the nodes where an initial copy of the mobile agent is loaded). The next places are visited by the mobile agent without any help from a service located within the cloud.

The **AC service** is activated when the cloud receives a *Req_AC* message from an agent (or from the AS service). As shown in Section 7.3, an agent migrates in the ad-hoc network until it reaches a place where its trip ends: the agent's status is either *End*, *Stop* or *Double*. Any *Req_AC* message contains the identities of the visited places. This information denotes either a complete itinerary (composed of one place per stage) or just a prefix of an itinerary. During the migration of an agent, a visited place only confirms that its current state allows to satisfy the request corresponding to the current stage. No binding booking is made at that time.

The AC service constructs possible itineraries and validates one of them when this is possible. The AS service and the monitoring tasks performed by each visited place ensure that at least one message will be received. Thus, AC does not require a timeout mechanism, as it can be sure that at least one agent will contact it. The AC service is activated when a first message related to a given transaction is received. The timeout Δ defined by the client, is just used to wait for possible additional messages. As each message may contain alternative places, this information can be used to replace invalid places in a previously evaluated itinerary. After the receipt of a *Req_AC* message, AC builds an itinerary (or at least the highest prefix) and tries to test it. All the places on this itinerary participate to the distributed atomic transaction. The AC service sends an *AskPlace* query to all the places and waits for their replies (*Yes* or *No*). A place that replies *Yes* must take the necessary steps for committing or aborting the transaction: it executes the transaction up to the point where it will be asked to commit or to abort. Based on the gathered votes, AC decides whether to commit (only if all votes are *Yes*) or to wait for alternative places that may substitute a place that voted *No* or has not yet replied. When the timeout Δ expires, the AC service aborts the transaction.

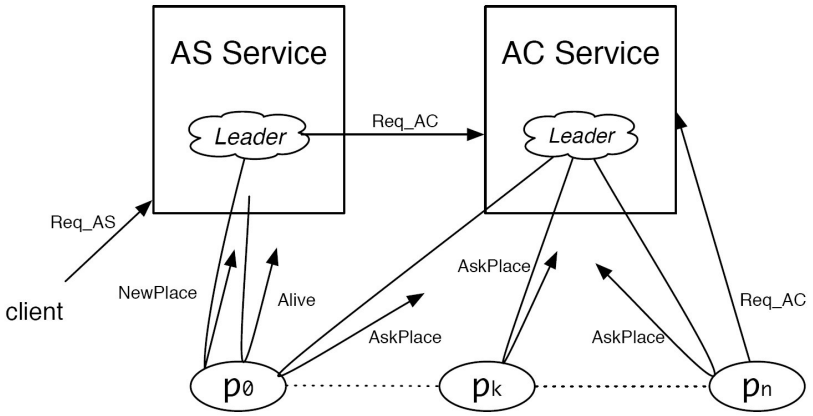


Figure 7.3: How the cloud and places interact.

The AC and AS services are independent from each other. When several transactions are executed simultaneously, these services run concurrently. Yet, for any transaction T_a, considered in isolation from the others, these services are used sequentially. As soon as at least one (complete or partial) itinerary is known by the AC service, the availability of a source has no more to be ensured. Conversely, while no itinerary is known, the validation

problem does not arise. As the services do not interfere, they can share a single data structure and just one record per transaction is used to store all the information. Figure 7.3 depicts the interaction between the ad-hoc network and the cloud.

7.5 A Unified Approach Based on Agreement

A single reliable machine in the cloud can act as a stable coordinator and provide both services. To cope with failures that may occur in the cloud, replicas have to be defined and a leader has to be elected among them. In this paper, each important act of the leader is the subject of a decision involving all copies. More precisely, a modification of the set of sources, a choice of the next itinerary to evaluate or a evaluation of the current itinerary require an agreement among all possible leaders. Due to the unique sequence of decisions, any new leader can restart in a consistent state.

The Paxos-MICprotocol is used to implement the functionalities of the agreement service, denoted by **AG**. Both the AS and the AC modules interact with the AG service through an intermediate level, called *Proposer Learner Dispatcher* and denoted by **PLD**. A PLD, an AS and an AC modules are attached to each coordinator. These modules interact locally and never communicate with remote nodes. By construction, only the modules associated to the leader are active. The general architecture used in the cloud is depicted in Figure 7.4. The communication messages between the AG level and the AS/AC modules pass in transit through the PLD. Any *ProposePull* call initiated by AG is first received at the PLD level of the leader. When the *ProposePull* function is invoked, the last decision value is attached and provided only to the PLD module of the current leader, which is acting as a unique learner. The *ProposePull* function is used both to provide the decision for the last consensus instance and to ask for a new proposal that will be used during the next consensus instance.

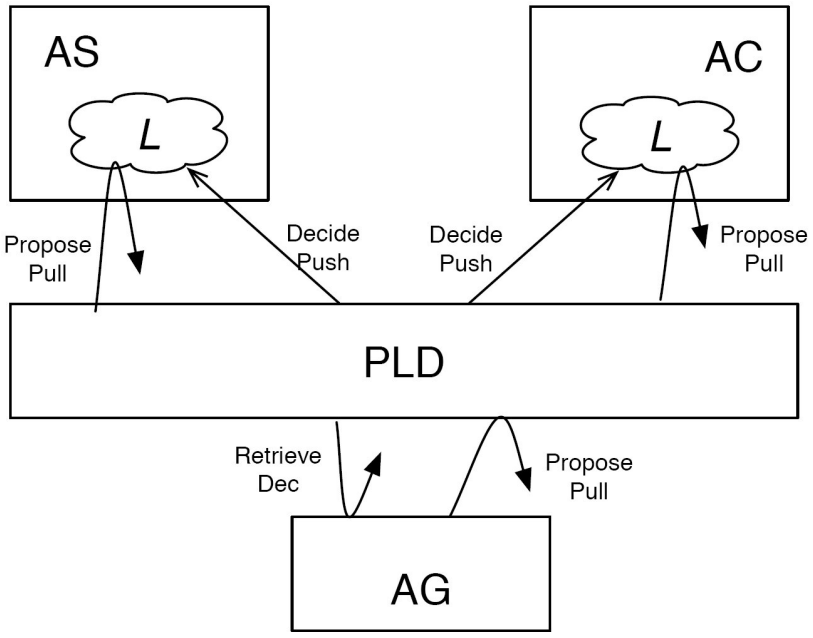


Figure 7.4: Architecture of the cloud services.

To limit the repeated use of the AG service, each proposal (decision) is the concatenation of several "small" proposals ("small" decisions), each referring only to one transaction. In this way, several transactions may be treated simultaneously during a single consensus instance. A small proposal or a small decision related to a transaction T_a, includes the identifier of T_a and a *Type* field that indicates the service currently provided for T_a. This structure is only relevant at the PLD and the AS/AC levels. The AG service does not require to know the structure of the value it decides upon. When the PLD receives a *ProposePull* call, it first checks if it already knows the complete sequence of previous decision values. This is not necessarily the case if the coordinator became the leader recently. If past decisions are missing, a call to the *RetrieveDec* function is addressed by the PLD level to the AG service. Any decision received by the PLD module, is split into small decisions that are routed by PLD to the corresponding service, by invoking the *DecidePush* function. In a second step, the *ProposePull* call is forwarded to both the AS and AC modules. The returned value also comes in transit through the PLD that acts as a proposer for the AG module. If we consider all the small decisions related to a transaction T_a, this sequence is composed of two parts. The prefix contains small decisions (at least one) related to the AS service. Each of these small decisions corresponds to an increasing set of monitored source places. By construction, the cardinality of this set increases by one with each taken decision. The last part is a subsequence of decisions concerning the AC service. Once an itinerary (complete or not) is learned, the AC service proceeds in constructing and evaluating itineraries. Two types of decisions are used. First, AC decides on the constructed itinerary: a small decision is a list of places. Then, AC decides on the outcome of the evaluation of this itinerary: a small decision is a list of votes provided by the places of the evaluated path. Once an itinerary has been evaluated, the transaction can end (*Commit* or *Abort*) or AC can wait for new information to construct and evaluate new itineraries. A decided itinerary is always followed by the list of votes provided by the places of the decided path. The actual information is contained in the *Data* field and consists of either a list of places or a list of votes. To cope with the timeout Δ, a *StartTime* field is included in each small decision.

7.6 Implementing the Services

The roles defined in the agreement protocol are also used in the implementation of the services. Coordinators and acceptors are also in charge of implementing the services provided by the cloud. Any request to activate either one of the services (*Req_AS* or *Req_AC*) is sent by a client or an agent to an available acceptor that forwards a message (*Monitor* or *Path*) to its leader. The requests pass in transit through the acceptors as only they are aware of a leader's identity (not necessarily the right one). As we assume $2+1$ acceptors and $f+1$ coordinators, the probability of contacting a non-crashed acceptor is greater than of finding an available coordinator. We assume that an agent can acquire the information it needs to contact several acceptors within the cloud, until it finds a non-crashed one.

Based on the messages they receive, acceptors and coordinators maintain a local data structure called *TransactionsDirectory* and denoted by TD. All data structures managed by the entities are displayed in figure 7.5. Each time a coordinator/acceptor becomes aware of a new transaction, it will create a corresponding entry in TD. This entry is created by invoking the *Put* function during an execution of tasks $Task_{AS}A1$, $Task_{AS}C1$, $Task_{AC}A1$ and $Task_{AC}C1$.

```
% Transaction Descriptor: Desc ← (Stages, Code, Places_0, Δ)
% Tag: has replied Yes, has replied No, not tested yet
% Last: true if the place belongs to the last tested itinerary
Tag ∈ {Yes, No, NT};  Place = (Id, Tag, Last);
% Knowledge is either a list of places or a list of lists of places
% Type of Service or Type of the next decision
Type ∈ {AS, AC1, AC2};  Data ∈ {"list of places", "list of votes"}
dec ← (Type, Tid, StartTime, Data)
% Transactions Directory: keep useful information for each transaction
TD ← [(Tid, Desc, Type, Knowledge, Result, StartTime)];
TD ← ∅;  Val ← [dec];  LastCon ← 0; % Last consensus number
```

Figure 7.5: Data structures.

These tasks[1] are executed by an acceptor when it receives service requests from clients and agents and by a coordinator, upon the receipt of a message from an acceptor.

```
% A client sends a request to initiate the execution of a transaction to an acceptor
Task_AS A1:When A_i receives msg Req_AS(Desc, Tid) from Client
(1)   if (Get(Req_AS.Tid, _, _, _, _, _) = false) then
(2)     Put(Req_AS.Tid, Req_AS.Desc, AS, _, _, _);
(3)     send Monitor(Req_AS.Desc, Req_AS.Tid) to C_Lid;

Task_AS C1:When C_i receives msg Monitor(Desc, Tid) from A_j
(4)   if (Get(Monitor.Tid, _, _, _, _, _) = false) then
(5)     Put(Monitor.Tid, Monitor.Desc, AS, _, _, _);
```

Figure 7.6: AS service: Monitor the first place.

Regarding a transaction T_a, the information logged in TD consists of several fields. The transaction identifier denoted by *Tid* and the transaction descriptor, *Desc*, store some static informations about T_a. The service currently provided for T_a by the cloud is indicated by the *Type* field. When the provided service is AC, this field also indicates the type of the next proposal the AC service must provide, regarding T_a, during a call to the *ProposePull* function. This proposal may carry a list of places ($Type = AC1$) or a list of votes ($Type = AC2$). All the useful information related to T_a is logged in a special field called *Knowledge*, maintained both by a coordinator and an acceptor. In the case of the AS service, this field is defined by the list of source places currently monitored. Regarding the AC service, the *Knowledge* is implemented as a list of lists of places: for each stage i of the mobile agent execution, $Knowledge[i]$ represents the list of alternative places where stage i of T_a was executed by different agents. Each acceptor periodically sends its current knowledge to its chosen leader (Figure 7.7, *TaskA*2), using *Monitor* or *Path* messages. Coordinators and acceptors adopt the same behavior: they increase the knowledge with new alternative places, each time new itineraries are observed by an acceptor, in *Req_AC* messages (Figure 7.7, $Task_{AC}A1$) and by a coordinator, in *Path* messages sent by acceptors (Figure 7.7, $Task_{AC}C1$).

The *IncreaseKnowledge* function tries to expand the current knowledge with new re-

[1] In the notation, AS/AC identifies the service, while Ax/Cx distinguishes acceptors' tasks from coordinators' tasks.

```
Task_AC A1: When A_i receives msg Req_AC(Tid, State, Itinerary)
from Agent_k
(1)   if (Get(Req_AC.Tid, _, Type, K, _, _) = false) then
(2)     IncreaseKnowledge(K, Req_AC.Itinerary);
(3)     Put(Req_AC.Tid, _, AC1, K, _, _);
(4)   else if (Type = AS) then K ← ∅; Type ← AC1;
(5)     IncreaseKnowledge(K, Req_AC.Itinerary);
(6)     Update(Req_AC.Tid, _, Type, K, _, _);
(7)   send Path(Req_AC.Tid, K) to C_Lid;

TaskA2: Periodically
(8)   for each (i < TD.size()) do
(9)     if (TD[i].Type = AS) then
(10)      send Monitor(TD[i].Tid, TD[i].Knowledge) to C_Lid;
(11)    else
(12)      send Path(TD[i].Tid, TD[i].Knowledge) to C_Lid;
```

Figure 7.7: A_i: validate the best path.

```
Task_AC C1: When C_i receives msg Path(Tid, K) from A_j
(1)   if (Get(Path.Tid, _, Type, K, _, _) = false) then
(2)     Put(Path.Tid, _, AC1, Path.K, _, _);
(3)   else if (Result ≠ ⊥) then
(4)         if (Type = AS) then K ← ∅; Type ← AC1;
(5)         IncreaseKnowledge(K, Path.K)
(6)         Update(Path.Tid, _, Type, K, _, _);
```

Figure 7.8: C_i: validate the best path.

ceived information. The outcome of the validation process for T_a is stored in the *Result* field. As the computation process progresses, different places in the knowledge are tested in order to construct the best path. Testing a place p_i for a transaction T_a consists in obtaining a vote (*Yes* or *No*) from p_i regarding T_a. A coordinator must be able to distinguish between a place not yet tested and a tested place that has replied *Yes* or *No*. Along with the identifier of a place p_i, a coordinator logs in its knowledge, a tag that indicates if p_i was tested or not, denoted by *Tag*, and also a boolean field, *Last*, set to *true* if p_i was included in the last tested path. Any PLD module also maintains information related to decision values sent by the AG level, such as the last completed consensus instance, *LastCon*.

The execution of $Task_{PLD}C1$, in Figure. 7.9 is performed only by the PLD of the current leader. It is triggered by an invocation to the *ProposePull* function. The leader is invited to propose a value for the current consensus instance denoted by the variable *Con*. Moreover, this function provides to the leader, the last known decision (variable *DVal*). In order to construct a new proposal, the leader must acquire a complete knowledge of the current stage of the validation process. Thus, it first retrieves all missing decisions, related to consensus instances i, with $LastCon < i < Con - 1$.

The past decision values are routed through the PLD level towards the AS or AC service, according to the service type of the decision. For a transaction T_a, the leader will execute either $Task_{AC}C2$ or $Task_{AS}C2$, described in Figure. 7.10. The retrieved decision is

```
Task_PLD C1: Upon invocation of ProposePull(Con, DVal)
returns PVal
(1)  for each (i = LastCon + 1; i< (Con − 1); i++) do
(2)      Val ← RetrieveDec(i);
(3)      for each (dec ∈ Val) do
(4)          if (dec.Type = AS) then DecidePush_AS(dec);
(5)            else DecidePush_AC(dec);
(6)  if (DVal.Type = AS) then DecidePush_AS(DVal);
(7)    else DecidePush_AC(DVal);
(8)  LastCon ← Con-1;
(9)  PVal_1 ← ProposePull_AS(); PVal_2 ← ProposePull_AC();
(10) PVal ← PVal_1 ∪ PVal_2;
(11) return PVal;
```

Figure 7.9: Relay ProposePull.

used by a leader to update its current knowledge. For the AS service, the leader just adds the newly observed source places to its knowledge. For the AC service, the leader first checks the type of the information contained in the value. If the decision carries a list of votes, the leader computes the outcome of the currently considered path. If the result returned by the *CheckList* function is *Commit* (and if the tested itinerary is a complete one, *i.e.* its size is equal to the number of stages), the validation process for T_a is completed and the client is notified of the constructed path. Otherwise, the leader uses the retrieved votes to filter its current knowledge.

```
Task_AC C2: Upon invocation of DecidePush_AC(dec)
(1)  Get(dec.Tid, Desc, Type, K, Res, _);
(2)  if (Res ≠ ⊥) then STOP;
(3)  if (dec.Type = AC2) then
(4)    if (dec.StartTime ≠ 0) then
(5)      Update(dec.Tid, _, _, _, _, dec.StartTime);
(6)    Res ← CheckList(dec.Data);
(7)    if ((Res = Commit) ∧ (dec.Data.size() = Desc.Stages)) then
(8)      Update(dec.Tid, _, _, _, Res, _);
(9)    else FilterKnowledge(dec.Data, K);
(10)       Update(dec.Tid, _, AC1, K, _, _);
(11) else IncreaseKnowledge(K, dec.Data); SetLast(dec.Data, K);
(12)   Update(Dec.Tid, _, AC2, K, _, _);

Task_AS C2: Upon invocation of DecidePush_AS(dec)
(13) Get(dec.Tid, Desc, _, K, _, _);
(14) IncreaseKnowledge(K, dec.Data);
(15) Update(dec.Tid, _, _, K, _, _);
```

Figure 7.10: Usage of a decision.

The filtering mechanism updates the tag of the last tested places in the knowledge, according to the vote provided by each of them. If the leader retrieved a list of places, this decision indicates the last tested path for T_a. This itinerary expands the current knowledge of the leader with new places, if any. After exploiting all the retrieved information, the

leader is now able to construct a proposal. The result of its computation will contain several proposals related to different transactions. Thus, services are provided simultaneously to several transactions.

```
Task_AC C3:Upon invocation of ProposePull_AC() returns PVal
(1)   PVal ← ∅;
(2)   for each (Tid ∈ TD) do
(3)       Get(Tid, Desc, Type, K, Res, ST);
(4)       if ((Res ≠ ⊥) ∧ (Type ≠ AS)) then
(5)           PVal←PVal ∪ NewProposal_AC(Type,Tid,Desc,K,ST);
(6)   return PVal;

Task_AS C3: Upon invocation of ProposePull_AS() returns PVal
(7)   PVal ← ∅;
(8)   for each (Tid ∈ TD) do
(9)       Get(Tid, Desc, Type, _, Res, _);
(10)      if ((Res ≠ ⊥) ∧ (Type = AS)) then
(11)          PVal ← PVal ∪ NewProposal_AS(Tid, K, Desc);
(12)  return PVal;
```

Figure 7.11: Providing a proposal at the AC and AS Service modules.

For each transaction T_a, logged in the knowledge, the leader invokes the $NewProposal_{AS}$ or $NewProposal_{AC}$ function, according to the service provided for T_a (see Figure. 7.11). The $NewProposal_{AS}$ function constructs a new proposal for T_a only if all the monitored source places are suspected (see section 7.4). In the case of the AC service, the *Type* field of the knowledge indicates the type of the next proposal for transaction T_a. Two cases are possible. 1) If the last observed decision was a list of votes obtained from a path that did not successfully commit the transaction, the leader tries to compute a new path (function *GetNewPath*). During its research, the leader will explore the knowledge corresponding to T_a, with the purpose of finding useful alternative places (function *FindAlternative*). A place is useful if it was not tested or its vote was *Yes*. If this research fails, the leader waits (during a timeout specified by the user) for agents to arrive with new itineraries. Once the timeout for T_a expired, the leader will abort the transaction. 2) If the last decision for T_a was an itinerary, the new proposal must include the list of votes corresponding to the places of this itinerary. By invoking the *GatherVotes* function, the leader obtains from each place on the path, a vote, regarding T_a. After constructing the list of votes, the leader arms the timeout mechanism for T_a, by setting the *StartTime* to the current time. The client specifies through this timeout, the amount of time it is willing to wait for its transaction to be committed, if the first tested path yielded *Abort*. The *StartTime* for T_a is included in the new proposal that will become the next decision value. In this way, it will be observed by any future leader that will retrieve past decisions.

7.7 Evaluation

The code executed by a mobile agent was implemented on top of an existing mobile agent platform called *JADE* [1]. The *J*ava *A*gent *DE*velopment framework represents a software implemented in Java language that facilitates the implementation of multi-agent systems.

```
Function IncreaseKnowledge(Knowledge, List)
% List can be a list of places or a list of lists
(1)   for each (m_j ∈ List) do
(2)       Knowledge[j] ← Knowledge[j] ∪ ([m_j]);

Function NewProposal_AC(Type, Tid, Desc, K, ST) returns PVal
(3)   PVal ← ⊥;
(4)   if (Type = AC1) then
(5)     List ← GetNewPath(K);
(6)     if (List ≠ ⊥) then PVal ← (AC1, Tid, List);
(7)     else if ((ST + Desc.Δ) < CurrentTime) then
(8)             Update(Tid, _, _, K, Abort, _);
(9)   else List ← FindLastTested(K);
(10)      Votes ← GatherVotes(List, Tid);
(11)      if(ST = 0) then ST← CurrentTime;
(12)      PVal ← (AC2, Tid, ST, Votes);
(13)      Update(Tid, _, _, _, _, ST);
(14)  return PVal;

Function NewProposal_AS(Tid, K, Desc) returns PVal
(15)  PVal ← ⊥;
(16)  if (Alive(K) = False) then
(17)    p ← NewPlace(Desc.Places_0);
(18)    if (p = ⊥) then send Req_AC(Tid, Stop, ⊥) to A_j;
(19)    else K ← K ∪ [p];  PVal ← (AS, Tid, K);
(20)  return PVal;

Function FilterKnowledge(votes, K)
(21)  for each (i < K.size()) do
(22)      for each (p ∈ K[i]) do
(23)          if (p.Last = true) then p.Tag ← votes[i]); break;
```

Figure 7.12: Functions.

JADE offers a set of graphical tools for the debugging and deployment phases. The agent platform can be distributed across several sites and a graphical user interface (GUI) allows a remote control of the configuration.

The proposed mobile agent system was developed on top of *JADE*. The actual Java classes implementing the mobile agent code, are not available on the places.

The experimental settings consist of 6 machines with processors Pentium IV, 3 GHz and 1 GB of RAM, connected by a 100 Mb/s Ethernet network. The JADE platform is running under JDK1.5 Virtual Machine.

In a first testing scenario, the goal is to measure the time required by a mobile agent execution when the itinerary consists of an increasing number of stages. This experiment was carried out by using first a simple mobile agent and then, a fault-tolerant mobile agent. The obtained results are displayed in Figure 7.14.

An analysis of the figure, shows that the FT-Agent introduces a performance overhead of 160% compared to a simple mobile agent execution, without integrating fault-

```
Function GetNewPath(K) returns Places or ⊥
(1)   Places ← ⊥;
(2)   for each (i < K.size()) do
(3)       Places←Places ∪ FindAlternative(K[i]);
(4)   return Places;

Function SetLast(places, K)
(5)   ResetLast(K, false);
(6)   for each (i < K.size()) do
(7)       for each (p ∈ K[i]) do
(8)           if (p.Id = places[i])) then p.Last ← true; break;

Function CheckList(List) returns Abort ∨ Commit
(9)   for each (v ∈ List) do if (v = NO) then return Abort;
(10)  return Commit;

Function GatherVotes(List, Tid) returns Votes
(11)  for each (p_i ∈ List) do
(12)      vote←AskPlace(p_i, Tid);   Votes←Votes ∪ {vote};
(13)  return Votes;

Function FindAlternative(List) returns alt or ⊥
(14)  alt ← ⊥;
(15)  for each (p ∈ List) do
(16)      if (p.Tag = Yes) then return p;
(17)      else if (p.Tag = NT) then alt ← p;
(18)  return alt;

Function FindLastTested(K) returns List
(19)  List ← ⊥;
(20)  for each (i < K.size()) do
(21)      for each (p ∈ K[i]) do
(22)          if (p.Last = true) then List ← List ∪ [p]; break;
(23)  return List;
```

Figure 7.13: Functions.

tolerant mechanism. However, the protocol that renders the mobile agent fault-tolerant, provides two main advantages. First, a temporal replication provides fault-tolerance during the mobile agent trip. Second, the transactional scheme copes with the case of long-running transactions, in which the booking may be canceled before the end of the mobile agent trip. Thus, the whole transaction will not be committed. The proposed protocol anticipates the detection of this semantic failure by launching a new agent on another path before the end of the transaction.

In a second testing scenario, we aim at measuring the time required by the Atomic Commit phase implemented by relying on the Paxos-MIC protocol. For this purpose, we considered a mobile agent execution with a varying number of stages per itinerary. Figure 7.15 depicts the time required by the mobile agent execution in the ad-hoc network and the time required by the AC service to validate the transaction as a whole. The latency introduced by the AC service implemented by using Paxos-MIC as a building-block, does not

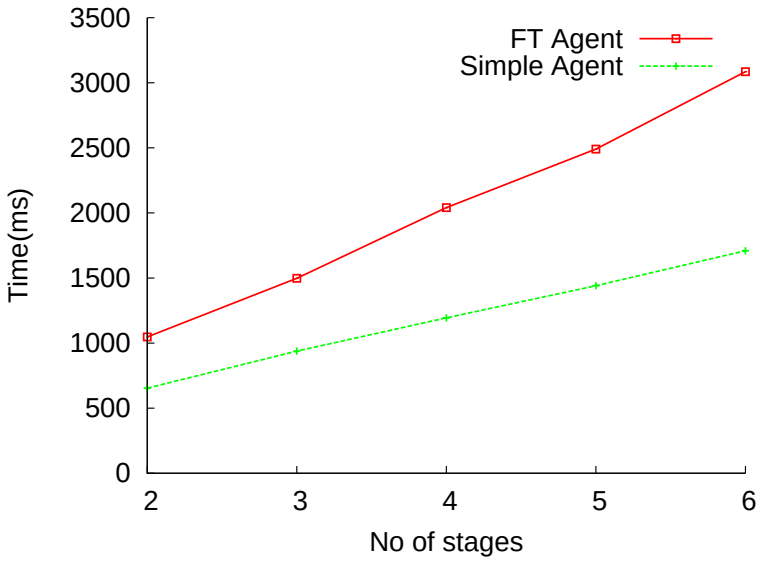


Figure 7.14: Time required by a simple agent and a FT-Agent execution.

introduce a significant overhead.

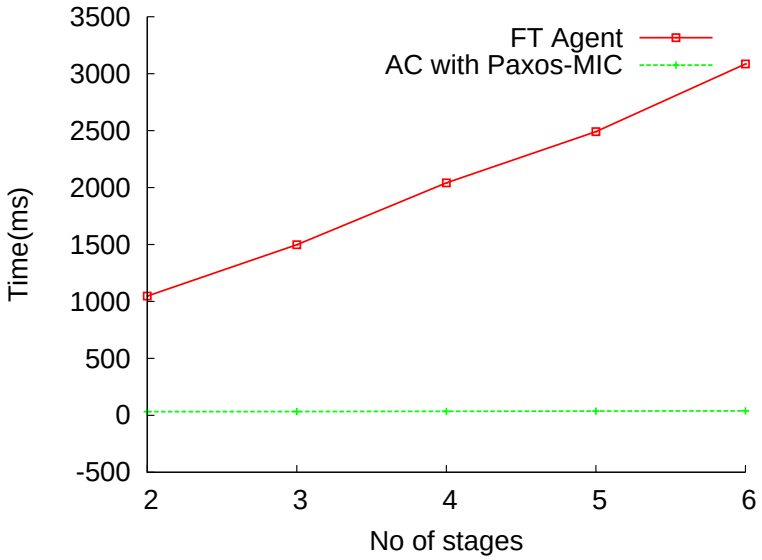


Figure 7.15: Time required by a FT-Agent execution and the AC protocol.

7.8 Final remarks

In this chapter, we focused on fault-tolerant distributed applications that rely on an everlasting sequence of decision values. In this general context, we addressed the concept of transactions, recognized as a powerful approach to model distributed applications. Further,

we considered the particular case of transactions executed in an ad-hoc network, by relying on a mobile agent system. We proposed a solution to support the execution of a transactional mobile agent, solution that provides two important services: the monitoring of the source places (AS service) and atomic commit of the transaction (AC service). In our approach, these services are made reliable by implementing them on top of a unique agreement service that generates a sequence of decisions. The agreement service is provided by the Paxos-MIC protocol, used as a consensus building-block.

Part V

Conclusions: Achievements and Perspectives

Chapter 8

Conclusions

Contents

THE Consensus problem is a central paradigm of fault-tolerant distributed computing, as it encapsulates many challenges that arise from the design reliable and efficient distributed applications. The work presented in this thesis represents a contribution to the understanding of these challenges and to building efficient distributed consensus protocols.

The work carried out throughout this PhD has led to the development of *Paxos-MIC*, an efficient consensus protocol that allows reaching fast and everlasting decisions. We motivated the need for such a protocol by arguing the importance of generating a persistent sequence of decision values. Such a sequence lies at the heart of many important fault-tolerant techniques, the *state machine* approach representing a reference argument in this concern. We evaluated the main features of the Paxos-MIC protocol through extensive experimentations and also focused on identifying contextual factors that impact the performance of the framework.

In this chapter, we resume our contributions and outline some perspectives for future directions.

8.1 Achievements

In the context of asynchronous distributed systems prone to crash failures, providing efficient solutions to agreement problems is a key issue when designing fault-tolerant applications. Our core contribution was the design and evaluation of a protocol able to efficiently

build an everlasting sequence of decisions. Another major contribution studied the use of such a sequence of decisions as support for designing higher-level applications. In the following, we provide a summary of our contributions.

Formal definition of the Multiple-Integrated Consensus problem. We defined the *Multiple-Integrated Consensus* problem that allows us to identify the context of our approach and the motivation for proposing our protocol: the focus on both the construction and the availability of a sequence of decision values. We extended the classical formal definition of the consensus problem in order to specify properties when coping with a sequence of consensus instances.

Paxos-MIC: an adaptive fast Paxos for making quick everlasting decisions. We proposed the *Paxos-MIC* protocol, an efficient framework that is adaptive, reaches fast decisions and ensures the persistence of all decision values. The Paxos-MIC protocol allows to solve a sequence of consensus instances in an unreliable, asynchronous system. At this algorithmic level, our contributions were defined in several steps. We revisited the interaction scheme between proposers, learners, coordinators and acceptors, based on which we constructed an architecture that describes the communication pattern between all involved entities. Further, we provided an algorithmic description of the Paxos-MIC protocol, sustained by a detailed description of the underlying principles of the protocol. The key targets of the protocol's design were efficiency and adaptability. Indeed, Paxos-MIC integrates two optimizations: a safe optimization (denoted $\mathbf{S_O}$) that is always activated and a risky optimization (denoted $\mathbf{R_O}$) that is activated at runtime, only in favorable circumstances. The main feature of the framework is its *adaptability*: for each consensus instance, the leader checks at runtime if the context seems suitable for activating optimization $\mathbf{R_O}$. A detailed proof of a protocol similar to Paxos-MIC can be found in [32].

Interest of activating the risky optimization. An important focus of our work was optimization $\mathbf{R_O}$. As this optimization may be counterproductive, when used in unfavorable circumstances, we investigated the conditions that enable a performance gain when optimization $\mathbf{R_O}$ is activated. To meet this goal, we performed extensive synthetic benchmarks using Paxos-MIC, with two major goals: to analyze the impact of some contextual factors (size of the core, geographical position of the actors) on the time required to reach a decision and also to obtain an assessment of the performance degradation of optimization $\mathbf{R_O}$ when used in less favorable circumstances.

Prediction of collisions. Optimization $\mathbf{R_O}$ succeeds only in favorable conditions: all proposers provide the same initial value for a given consensus instance. Otherwise, a *collision* occurs and the optimization may lead to a significant additional cost. An important challenge for our work was to determine if collisions can be predicted in a reliable manner. For this purpose, we proposed and evaluated several *triggering criteria* used by the leader to decide the activation of optimization $\mathbf{R_O}$, at runtime and depending on the current context. The definition of these criteria relied on different knowledge, such as an analysis of the recent past, the study of the current context and also a possible prediction of the future. We evaluated the efficiency of the criteria by considering a particular application (a secure

Web server) and a real trace that records the activation dates of the successive consensus instances. Through an analysis of the trace, we measured the expected gain for each of the defined criterion and we observed the accuracy of the triggering criteria in predicting future collisions.

Consensus as a building block. Many fault-tolerant distributed applications are built upon a consensus service in charge of generating a sequence of decision values. Considering this as a motivation, we addressed the context of transactional mobile agents and proposed a solution to support the execution of transactions in ad-hoc networks. We defined two important services that provide support for the agent execution and showed how these services can be implemented by relying on an everlasting sequence of decisions. This sequence was generated by the Paxos-MIC protocol. To guarantee reliability, our solution relies on a single agreement protocol that orderes continuously all the new actions whatever the related transaction and service. This work was carried out in collaboration with Linda Zeghache and Nadjib Badache from CERIST, Algeria.

Implementation and evaluation. An important part of this work has been dedicated to providing an efficient practical implementation of the Paxos-MIC protocol, based on the algorithmic description. A great amount of work has also been invested in the evaluation of the protocol, through a series of synthetic benchmarks. All experiments involved in the aforementioned contributions were carried out on the Grid'5000/ALLADIN experimental testbed federating 10 different sites in France. It is an initiative of the French Ministry of Research through the ACI GRID incentive action, INRIA, CNRS, RENATER and other contributing partners. We are particularly grateful for the excellent support that was provided by the Grid'5000 team during the time in which the work presented in this thesis was carried out.

8.2 Perspectives

The research described in this thesis opens several perspectives that we find interesting to pursuit. In this section, we discuss future directions.

Paxos-MIC for byzantine systems. Paxos-MIC was designed to solve consensus in a crash-failure system model in which a process crashes by halting its execution. An interesting future direction is the extension of Paxos-MIC in order to support *byzantine* failures. In a byzantine failure model, processes can fail arbitrarily: failing to send or receive a message, corrupting local state, or sending an incorrect reply to a request. Optimization $\mathbf{S_O}$ for the byzantine model was already proposed in [44]. The Paxos approach in byzantine model requires $5f + 1$ acceptors, $3f + 1$ proposers and $3f + 1$ learners, where f represents the maximum number of byzantine failures in the system. There are several challenges that arise when considering Paxos-MIC for the byzantine model. As processes may have a malicious behavior, important actions should be made only after "enough" knowledge has been gathered. For instance, each coordinator updates its tag according to the most recent information observed in the *State* messages received from acceptors. To tolerate malicious behavior, this

update of the tag should be made only after the coordinator has observed a higher tag in "enough" messages sent by acceptors. With regard to optimization $\mathbf{R_O}$, a malicious proposer may provide different values to different acceptors, thus generating collisions on purpose. However, given the functioning mechanisms of the Paxos-MIC protocol, such a behavior would postpone the decision, but not prevent it. Once a leader observes a collision, it begins the recovery procedure by initiating a *Prepare* phase. After this phase, the leader will select the most frequent value among the ones gathered and will attempt to decide on this value.

Consensus in anonymous asynchronous systems. In the context of distributed computing, many challenges arise as a result of asynchrony and failures. In a recent past, a great part of specialized literature has focused on a new uncertainty aspect, namely *anonymity*. This concept introduces new challenges in distributed computing. Roughly speaking, this means that, apart from being asynchronous and prone to crash failures, processes have no identity and consequently, they cannot distinguish the ones from the others. Anonymity represents a challenge of great interest for domains in which guaranteeing privacy becomes a crucial issue. In an anonymous system, processes have no name and execute the same algorithm. Solving consensus in such a system, becomes even more difficult. A recent work has focused on defining failure detectors for the anonymous system model [10]. The authors propose several classes of anonymous failure detectors that represent the counterparts of already existing classes in the classical system model. In [24], the authors investigate under which conditions information can be reliably shared and consensus can be solved in unknown and anonymous message-passing networks prone crash-failures. To this goal, they define some synchrony assumptions that extend the anonymous system and render consensus solvable.

We have recently directed our research towards solving consensus in anonymous systems. Most of the algorithms for distributed systems consider that the number of processes in the system is known and every process has a unique identifier. However, in some networks such as in wireless sensors networks, this is not necessarily true. The lack of identities leads to many challenges, some of which are already under study.

Different applications, same building block. Many fault tolerant techniques are implemented by relying on a unique and everlasting sequence of decisions. Such a sequence is usually constructed by invoking repeatedly a consensus service provided by a dedicated set of n nodes. Although the service provided by the consensus building-block is the same, the upper-layer applications interact differently with this building-block. Many works have been devoted to optimizing the performance of the consensus protocol used as a building-block. However, little focus has been dedicated to the interaction between the building-block and the application itself. This interaction differs according to the application's requirements. If we consider a consensus building-block capable of constructing a sequence of decisions, is this block "plugable" in any application's architecture? A possible future direction would be to identify various factors that define the interaction between a consensus building-block and an upper-layer application. Considering Paxos-MIC as an engine generating a sequence of persistent decisions, an interesting challenge would be the construction of an interface versatile enough to connect such a building block with different applications with various requirements.

A practical estimation of the risk of collisions. In chapter 6, part of the work has focused on collisions and the possibility to predict them. We evaluated the accuracy of different triggering criteria in four different contexts by analyzing the logs of a real application: a real trace that records the activation dates of the successive consensus. In all our measurements, we used real values for durations of consensus, values obtained as a result of several experimentations. For instance, D_{norm}, D_{succ} and D_{fail} have been obtained through several experiments. However, for the function that estimates the collisions risk, we have no practical support. We chose several definitions for this estimation. The definition we used for the measurements considers that a collision occurs during the consensus instance c if $\mathbf{R_O}$ is activated during this consensus and at least another proposal is generated before the end of the consensus. Two other weaker definitions of the risk have been studied. In both, the risk of collision is no more the same during the whole execution of the consensus instance. In our second estimation of the risk, we consider that a risk exists only during the first half of the execution and is null after. In our third estimation of the risk, we consider that the level of risk decreases uniformly throughout the execution.

During our experiments, we made several attempts at obtaining a real definition of this risk estimation. We replayed the sequence of logs in both local areas and wide areas settings with the purpose of observing how often collisions appear. However, in real settings, there are many issues that may appear and are out of the control of the developer. In particular, synchronizing threads on multi-core machines is one of the hardest to manage challenge. In addition, message losses in local area networks, increase the probability of collisions. On the contrary, network devices such as routers or switches, create an order on the packages they forward, which reduces the risk of collisions.

Amnesic logs. Decision values generated by the Paxos-MIC protocol, are *p*ersistent. The logging mechanism implemented by acceptors, ensures that any past decision value can be retrieved by any interested learner at any time. The logs can also ensure the termination property stating that, during each consensus instance, at least one correct process eventually decides a value. However, in an asynchronous system, these logs might store an un-bounded number of values. If weaker termination properties are considered, it is possible to implement amnesic logs that store only a limited number of decision values [20]. Paxos-MIC can be extended to integrate a mechanism that "cleans" the logs after a specific time, by erasing the entries that contain decisions considered as being "old". Classifying a decision values as being "ol'd' depends on the application built on top of the sequence of decisions. We may imagine following the model of the garbage collection mechanisms to implement these amnesic logs. Each decision value has associated a counter indicating the number of times the value was retrieved, at the request of a learner. When this counter is not being updated for a given timeout, we may assume that the corresponding value is no longer requested by learners and may be deleted from the logs.

Bibliography

[1] The Java Agent DEelopement framework. http://jade.tilab.com/.

[2] The Kadeploy project. http://kadeploy.imag.fr/.

[3] The OAR project. http://oar.imag.fr/.

[4] The Writings of Leslie Lamport. http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html.

[5] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing omega with weak reliability and synchrony assumptions. In *PODC'03*, pages 306–314, 2003.

[6] A. Basu, A. Lin, and S. Ramanathan. Routing using potentials: a dynamic traffic-aware routing algorithm. In *Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications*, SIGCOMM '03, pages 37–48, New York, NY, USA, 2003. ACM.

[7] M. Ben-Or. Another advantage of free choice: completely asynchronous agreement protocols. In *Proc. ACM Symposium on Principles of Distributed Computing*, pages 27–30. ACM Press, 1983.

[8] R. Boichat, P. Dutta, S. Frolund, and R. Guerraoui. Deconstructing Paxos. *ACM SIGACT News*, 34(1):47–67, 2003.

[9] R. Boichat, P. Dutta, and R. Guerraoui. Asynchronous leasing. *IEEE Int. Workshop on Object-Oriented Real-Time Dependable Systems*, 2002.

[10] F. Bonnet and Michel Raynal. Anonymous asynchronous systems: the case of failure detectors. In *Proceedings of the 24th international conference on Distributed computing*, DISC'10, pages 206–220, Berlin, Heidelberg, 2010. Springer-Verlag.

[11] E. Borowsky and E. Gafni. Generalized flp impossibility result for t-resilient asynchronous computations. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, STOC '93, pages 91–100, New York, NY, USA, 1993. ACM.

[12] F. Brasileiro, F. Greve, M. Hurfin, J.P. Le Narzul, and F. Tronel. Eva: an Event-Based Framework for Developing Specialised Communication Protocols. *In IEEE Int. Symp. on Network Computing and Applications*, pages 108–119, 2002.

[13] L. J. Camargos, R. M. Schmidt, and F. Pedone. Multicoordinated agreement protocols for higher availabilty. *NCA, Proc. of the 7th IEEE Int. Symp. on Networking Computing and Applications*, pages 76–84, 2008.

[14] F. Cappello, E. Caron, M. Dayde, F. Desprez, E. Jeannot, Y. Jegou, S. Lanteri, J. Leduc, N. Melab, G. Mornet, R. Namyst, P. Primet, and O. Richard. Grid'5000: A large scale, reconfigurable, controlable and monitorable grid platform. In *Grid '05: Proc. 6th IEEE/ACM Intl. Workshop on Grid Computing*, pages 99–106, Seattle, Washington, USA, November 2005.

[15] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43:685–722, July 1996.

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

[17] B. Charron-Bost and A. Schiper. The Heard-Of Model: Unifying all Benign Failures. Technical report, 2006.

[18] B. Charron-Bost and A. Schiper. Improving fast paxos: being optimistic with no overhead. *PRDC, 12th IEEE Pacific Rim Int. Symp. on Dependable Computing*, pages 287–295, 2006.

[19] S. Chaudhuri. Agreement is harder than consensus: set consensus problems in totally asynchronous systems. In *Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, PODC '90, pages 311–324, New York, NY, USA, 1990. ACM.

[20] G. Chockler, R. Guerraoui, and I. Keidar. Amnesic distributed storage. *Proc. of the 21st Int. Symp. on Distributed Computing (DISC'07)*, pages 139–151, 2007.

[21] B. Claudel, G. Huard, and O. Richard. TakTuk, adaptive deployment of remote executions. In *HPDC '09: Proceedings of the 18th ACM international symposium on High performance distributed computing*, pages 91–100, New York, NY, USA, 2009. ACM.

[22] S. Czerwinski, B. Zhao, T. Hodes, A. Joseph, and R. Katz. An architecture for a secure service discovery service. In *Proceedings of the 5th annual ACM/IEEE international conference on Mobile computing and networking*, MobiCom '99, pages 24–35, New York, NY, USA, 1999. ACM.

[23] C. Delporte-Gallet, S. Devismes, H. Fauconnier, F. Petit, and S. Toueg. With finite memory consensus is easier than reliable broadcast. *OPODIS, 12th Int. Conf. on Principles of Distributed Systems*, 5401:41–57, 2008.

[24] C. Delporte-Gallet, H. Fauconnier, and A. Tielmann. Fault-tolerant consensus in unknown and anonymous networks. *2009 29th IEEE International Conference on Distributed Computing Systems*, pages 368–375, 2009.

[25] D. Dobre, M. Majuntke, M. Serafini, and N. Suri. Hp: Hybrid Paxos for WANs. *European Dependable Computing Conference*, pages 117–126, 2010.

[26] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony (preliminary version). In *Proceedings of the third annual ACM symposium on Principles of distributed computing*, PODC '84, pages 103–118, New York, NY, USA, 1984. ACM.

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

[28] S. Frolund and R. Guerraoui. Implementing e-transactions with asynchronous replication. *IEEE Trans. Parallel Distrib. Syst.*, 12(2):133–146, 2001.

[29] S. Frolund and R. Guerraoui. e-transactions: End-to-end reliability for three-tier architectures. *IEEE Trans. Software Eng.*, 28(4):378–395, 2002.

[30] R. Guerraoui and M. Vukolic. Refined quorum systems. *Distributed Computing*, pages 1–42, 2010.

[31] M. Hurfin, J.P. Le Narzul, F. Majorczyk, L. Mé, A. Saidane, E. Totel, and F. Tronel. A dependable intrusion detection architecture based on agreement services. *Proc. of the 8th Int. Symp. on Stabilization Safety and Security*, pages 378–394, November 2006.

[32] M. Hurfin and I. Moise. A multiple integrated consensus protocol based on Paxos, FastPaxos and Fast Paxos. *IRISA Technical Report, PI-1941*, Dec 2009.

[33] M. Hurfin, I. Moise, and J.P. Le Narzul. An adaptive Fast Paxos for making quick everlasting decisions. *The 25th IEEE Int. Conf. on Advanced Information Networking and Applications (AINA-2011)*, pages 208–215, March 2011.

[34] Y. Jégou, S. Lantéri, J. Leduc, Melab N., G. Mornet, R. Namyst, P. Primet, B. Quetier, O. Richard, E.G. Talbi, and T. Iréa. Grid'5000: a large scale and highly reconfigurable experimental grid testbed. *International Journal of High Performance Computing Applications*, 20(4):481–494, November 2006.

[35] D.B. Johnson. Routing in ad hoc networks of mobile hosts. In *Mobile Computing Systems and Applications, 1994. Proceedings., Workshop on*, pages 158 –163, dec 1994.

[36] I. Keidar. Timeliness, failure-detectors, and consensus performance. In *In PODC*, pages 169–178. ACM Press, 2006.

[37] L. Lamport. The Part-Time Parliament. *ACM Transaction on Computer Systems*, 16(2):133–169, May 1998.

[38] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):51–58, Dec. 2001.

[39] L. Lamport. Fast Paxos. *Distributed Computing*, 19(2):79–103, 2006.

[40] B. Lampson. The ABCDs of Paxos. *Proc. of the 20th Annual ACM Symp. on Principles of Distributed Computing*, 2001.

[41] V. Lenders, M. May, and B. Plattner. Service discovery in mobile ad hoc networks: A field theoretic approach. In *Proceedings of the Sixth IEEE International Symposium on World of Wireless Mobile and Multimedia Networks*, WOWMOM '05, pages 120–130, Washington, DC, USA, 2005. IEEE Computer Society.

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

[43] J. P. Martin and L. Alvisi. Fast byzantine consensus. *Proc. of the Int. Conf. on Dependable Systems and Networks*, pages 402–411, June 2005.

[44] J.P. Martin and L. Alvisi. Fast byzantine consensus. In *IEEE Transactions on Dependable and Secure Computing*, pages 402–411, 2005.

[45] A. Noor Mian, R. Baldoni, and R. Beraldi. A survey of service discovery protocols in multihop mobile ad hoc networks. *IEEE Pervasive Computing*, 8(1):66–74, 2009.

[46] A. Mostefaoui, S. Rajsbaum, and M. Raynal. Conditions on input vectors for consensus solvability in asynchronous distributed systems. *J. ACM*, 2003.

[47] A. Mostefaoui, S. Rajsbaum, M. Raynal, and M. Roy. Condition-based consensus solvability: a hierarchy of conditions and efficient protocols. *Distributed Computing*, pages 1–20, 2004.

[48] A. Mostefaoui and M. Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. In *Proceedings of the 13th International Symposium on Distributed Computing*, pages 49–63, London, UK, 1999. Springer-Verlag.

[49] S. Pleisch and A. Schiper. Non-blocking transactional mobile agent execution. In *ICDCS*, pages 443–444, 2002.

[50] S. Pleisch and A. Schiper. Fault-tolerant mobile agent execution. *IEEE Transactions on Computers*, 52:209–222, 2003.

[51] S. Pleisch and A. Schiper. Approaches to fault-tolerant and transactional mobile agent execution—an algorithmic view. *ACM Comput. Surv.*, 36:219–262, September 2004.

[52] F. Quaglia and P. Romano. Reliability in three-tier systems without application server coordination and persistent message queues. In *SAC*, pages 718–723, 2005.

[53] M. O. Rabin. Randomized byzantine generals. In *Proceedings of the 24th Annual IEEE Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[54] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Comput*, pages 149–157, 1997.

[55] F.B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, December 1990.

[56] S. Shepler, B. Callaghan, D. Robinson, R. Thurlow, C. Beame, M. Eisler, and D. Noveck. Network file system (NFS) version 4 protocol, 2003.

[57] R. Sher, Y. Aridor, and O. Etzion. Mobile transactional agents. In *Distributed Computing Systems, 2001. 21st International Conference on.*, pages 73 –80, apr 2001.

[58] M. Strasser and K. Rothermel. Reliability concepts for mobile agents. *Int. J. Cooperative Inf. Syst.*, 7(4):355–382, 1998.

[59] E. Totel, F. Majorczyk, and L. Mé. Cots diversity based intrusion detection and application to web servers. 3858:43–62, 2005.

[60] G. M. D. Vieira and L. E. Buzato. On the coordinator's rule for fast paxos. *Information. Processing Letters*, pages 183–187, March 2008.

[61] W. Zhao, L. Moser, and P. M. Melliar-Smith. A reservation-based extended transaction protocol. *IEEE Trans. Parallel Distrib. Syst.*, 19(2):188–203, 2008.

Appendix

Résumé en français

Contents

Nous fournissons ici un résumé long du manuscrit de thèse rédigé en langue anglaise. sectionse

I.1 Introduction

Les systèmes répartis sont désormais omniprésents. Ils ont émergé pour deux raisons principales. Tout d'abord, les entités qui interagissent dans le cadre d'une application (ou les

données qui sont exploitées) sont souvent dispersées géographiquement, indépendantes et parfois mobiles: du fait de sa nature même, l'application est répartie. La répartition peut également être un choix. Ainsi pour satisfaire des contraintes en terme de sûreté de fonctionnement, un service critique peut être répliqué sur des sites distants. Les multiples systèmes répartis se différencient, entre autre, par le niveau de synchronie qu'ils offrent, par les modes de communications entre processus qui sont disponibles, et par les modèles de défaillances qui les caractérisent.

Nous considérons un système réparti où les nœuds communiquent par échange de messages et peuvent connaître des défaillances de type *panne franche* ou de type *omission* lors de l'envoi ou lors de la réception d'un message. Un nœud qui ne tombe jamais en panne est dit *correct*. Inversement, un nœud est *incorrect* s'il arrête prématurément et définitivement son exécution durant le calcul. Un message peut être dupliqué ou perdu mais n'est jamais altéré. Par ailleurs, le système considéré est asynchrone: il n'existe pas de référentiel de temps commun et les temps de transfert des messages sont non bornés (ou du moins, si des bornes existent, elles ne sont pas connues).

La conception d'un système réparti soulève de nombreux défis. L'un des principaux défis concerne la fourniture de mécanismes de coordination entre les différents composants du système. Ces mécanismes sont indispensables. Par exemple, dans le cas d'un mécanisme de réplication destiné à assurer la tolérance aux défaillances, une cohérence forte doit être maintenue entre les différentes copies d'un serveur critique. Beaucoup de services essentiels, tel que la diffusion atomique ou la validation atomique peuvent se réduire en partie à un problème d'accord entre composants. L'approche "State Machine" en est une illustration. Dans cette approche, les répliques d'un serveur critique doivent s'entendre pour définir une séquence unique de requêtes entrantes. La construction de cette séquence se fait généralement en appelant de façon répétée un service de consensus. La thèse présentée dans ce manuscrit se focalise sur le problème du Consensus qui est sous-jacent à beaucoup de problèmes d'accord.

Résoudre efficacement des problèmes d'accord tel que le *Consensus* est un challenge important. Ce problème dont la spécification est relativement simple a fait l'objet de nombreux travaux de recherche. Durant une instance de consensus (identifiée par un entier c), des valeurs initiales (potentiellement distinctes) peuvent être proposées par un ou plusieurs nœuds appelés des *auteurs de propositions*. Chaque valeur est communiquée par son auteur à un sous ensemble unique de n nœuds parfaitement identifiés au sein du système global. Ces nœuds particuliers, que nous appellerons les *acteurs*, sont en charge d'exécuter le protocole de consensus. Les n acteurs ont pour mission de générer une séquence de décisions et d'en assurer la persistance afin que tout nœud intéressé puisse prendre connaissance des valeurs déjà décidées. Les nœuds intéressés sont appelés les *apprenants*. Un nœud qui joue le rôle d'auteur durant une instance de consensus joue aussi très souvent le rôle d'apprenant durant cette même instance. Cependant, en toute généralité, un nœud peut ne jouer qu'un seul de ces deux rôles. Les nombres d'auteurs et d'apprenants ne sont pas nécessairement bornés. Auteurs et apprenants peuvent être externes au sous-ensemble des acteurs ou membres de celui-ci. Durant une instance de consensus c, les n acteurs coopèrent entre eux afin de déterminer la valeur qui sera adoptée comme nouvelle valeur de décision. Par définition, le protocole de consensus exécuté par les acteurs doit permettre de converger inéluctablement (propriété de terminaison) vers une valeur de décision unique (propriété d'accord uniforme) qui doit nécessairement être l'une des valeurs proposées (propriété de validité). La valeur

de décision est alors retournée vers tous les apprenants qui se sont fait connaître (*i.e.*, dans cette étude, vers tous les auteurs qui ont fourni ou fourniront une valeur initiale concernant cette instance de consensus).

Dans un environnement réparti asynchrone où au moins un processus peut être défaillant, il a été prouvé qu'aucun protocole déterministe ne peut résoudre ce problème [27]. Néanmoins, ce résultat d'impossibilité peut être contourné en s'appuyant sur un service (détecteurs de défaillances, élection de leader, ...) qui sera utilisé comme oracle par le protocole. De nombreux protocoles indulgents ont été proposés: ils ne remettent jamais en cause les deux propriétés de sûreté qui caractérisent le problème du consensus (accord et validité). Par contre, dans ces protocoles, la propriété de vivacité (*i.e.*, la propriété de terminaison) n'est satisfaite que si l'oracle utilisé offre inéluctablement un niveau minimum de qualité de service. Dans cette thèse, nous nous intéressons à des protocoles ayant recours à un service d'élection de leader. En théorie, l'oracle doit être un service d'élection de leader ultime: il existe un instant à partir duquel un acteur correct est considéré par tous comme étant le seul leader au sein du groupe d'acteurs. En pratique, le service d'élection de leader doit identifier un acteur correct et le désigner comme étant le seul leader possible durant un laps de temps suffisamment long pour que le protocole de consensus ait le temps de converger vers une valeur de décision. Durant cette période, à chaque consultation de l'oracle, c'est l'identité de ce leader unique et correct qui est retournée.

Le protocole Paxos est le plus connu des protocoles de consensus s'appuyant sur un service d'élection de leader. Ce protocole a été initialement présenté par Lamport dans [37] puis reformulé de manière plus conventionnelle dans [38]. Dans Paxos, la notion de quorum majoritaire [30] est un élément clef qui régit les collectes d'information effectuées par un acteur. Les nœuds corrects sont supposés être majoritaires au sein du groupe de n acteurs: si f est le nombre maximum de pannes pouvant affecter le groupe de n nœuds alors $f < n/2$. Le protocole Paxos identifie deux rôles distincts: le rôle de coordinateur et le rôle d'accepteur. Chaque acteur joue un ou deux rôles. Une majorité d'accepteur doit être correcte: le nombre d'accepteurs doit donc être supérieur ou égal à $2f + 1$. Au moins un coordinateur doit être correct: le rôle de coordinateur est joué par au minimum $f + 1$ acteurs. Une asymétrie est créée entre les coordinateurs grâce au service d'élection de leader. Lorsque la qualité des informations fournies par l'oracle est parfaite, un seul coordinateur agit en temps que leader et coordonne la prise de décision. Paxos partage de nombreuses similitudes avec les protocoles s'appuyant sur des détecteurs de défaillances de la classe $\Diamond\mathcal{S}$ [16]. Ainsi, la progression du calcul durant une instance de consensus est rythmée par la notion de tour. Un tour est une tentative pour converger vers une valeur de décision. Chaque tour est identifié par un numéro et est géré par un coordinateur unique: le lien entre un numéro de tour et l'identité du coordinateur est une information prédéfinie connue de tous les acteurs. Le protocole Paxos n'impose pas aux acteurs qui exécutent un tour r d'avoir auparavant participé aux $r - 1$ tours précédents. Ceci a une conséquence intéressante: le protocole Paxos n'exige pas d'avoir des canaux de communication fiables. Cet atout supplémentaire explique l'intérêt porté au protocole Paxos qui se révèle par ailleurs tout aussi performant que les autres protocoles de consensus cités.

A partir du protocole original, plusieurs variantes ont ensuite été proposées afin notamment d'en améliorer les performances. Même si le protocole est conçu pour pouvoir évoluer dans des circonstances défavorables, les conditions d'exécution sont la plupart du temps favorables et c'est donc dans ces circonstances (normales et fréquentes) qu'il faut

obtenir des gains de performance. Durant l'exécution du protocole Paxos, les conditions d'exécution sont favorables si les trois conditions suivantes sont réunies. Premièrement, il ne doit pas se produire de défaillance (pas de panne d'un nœud, pas de perte de message). Deuxièmement, le système doit se comporter comme un système synchrone (les temps nécessaires au transfert d'un message ou à l'exécution d'un pas de calcul sont correctement estimés). Troisièmement, l'oracle doit être fiable et certain (le leader est correct et stable). Lorsque ces trois conditions sont réunies, le comportement du protocole est prévisible et le nombre d'étapes de communication peut alors être inféré via une analyse statique du code. Cette analyse se focalise sur l'enchainement des messages qui font qu'une valeur initiale v proposée par un auteur va finalement devenir la valeur de décision diffusée à tous les auteurs. Une analyse statique de ce type permet d'associer à chaque protocole une mesure correspondant au nombre d'étapes de communication requises.

I.2 Contributions

Les principales contributions de cette thèse peuvent être résumées de la façon suivante:

- **Définition formelle du problème du Consensus multiple intégré**
  Nous nous intéressons au problème de la construction et de la disponibilité d'une séquence de valeurs de décision durant toute la durée d'un calcul. Nous définissons ce double défis comme étant le problème du Consensus multiple intégré. Cette séquence est crée, pas à pas, par un sous ensemble plutôt stable de noeuds appelé le coeur. Ces noeuds se chargent également d'assurer la disponibilité des valeurs de décision déjà calculées. Nous étendons la définition classique du problème du consensus pour tenir compte du fait que les instances de consensus s'enchaînent.

- **Proposition d'un protocole adaptatif pour prendre rapidement des décisions persistantes**

  Nous avons conçu et développé un protocole adaptatif qui permet de générer et de stocker efficacement des décisions prises durant une séquence d'instance de consensus. Le protocole proposé s'appelle Paxos-MIC. Comme le suggère son nom, il s'appuie sur les principes généraux du protocole Paxos. Les interactions entres les différents types d'entités sont revisitées. Pour accroître les performances, nous considérons deux optimisations déjà connues qui visent à améliorer la latence du protocole Paxos original en réduisant le nombre d'étapes de communication. Ces optimisations sont appelées respectivement $\mathbf{S_O}$ et $\mathbf{R_O}$. Ces notations font référence au fait que la première optimisation, $\mathbf{S_O}$, est *sûre* alors que la seconde, $\mathbf{R_O}$, est *risquée*. L'optimisation $\mathbf{S_O}$ a été suggérée par Lamport [37, 38] et adoptée par la suite dans plusieurs autres travaux [40, 43, 8]. Cette optimisation vise à supprimer des étapes de communication lorsque celles-ci sont inutiles. Alors que l'algorithme original nécessite 6 étapes de communication, Paxos combiné avec l'optimisation $\mathbf{S_O}$ ne nécessite plus que 4 étapes de communication lorsque les conditions d'exécution sont favorables. La seconde optimisation a été présentée par Lamport dans [39]. Dans cet article, Lamport propose un protocole appelé Fast Paxos qui intègre l'optimisation $\mathbf{R_O}$ et améliore les performances du protocole Paxos lorsque les conditions d'exécution et les conditions d'utilisation sont favorables. Les conditions d'utilisation font référence aux comportements des auteurs. De fait,

l'optimisation $\mathbf{R_O}$ est plus exigeante en terme de contraintes. Durant une instance de consensus, l'optimisation $\mathbf{R_O}$ sera forcement couronnée de succès si tous les auteurs qui y participent proposent la même valeur. Dans le cas contraire, si au moins deux participants proposent des valeurs distinctes, un phénomène appelé *collision* risque éventuellement de se produire et une procédure de recouvrement devient nécessaire. Le gain escompté par rapport à une exécution normale du protocole se transforme alors en un surcoût en temps d'exécution qui peut être conséquent.

L'optimisation $\mathbf{R_O}$ est compatible avec l'optimisation $\mathbf{S_O}$. Alors que l'optimisation $\mathbf{S_O}$ concerne la première phase du protocole Paxos (la phase dite de *préparation*), l'optimisation $\mathbf{R_O}$ se focalise sur la deuxième et dernière phase du protocole (la phase dite de *proposition*). L'optimisation $\mathbf{R_O}$ consiste, d'une part, à anticiper une partie du calcul (*i.e.*, à exécuter entièrement la phase de préparation et à débuter la phase de proposition avant même que des valeurs initiales soient proposées par des auteurs) et, d'autre part, à désigner les accepteurs comme étant les destinataires directs des valeurs initiales proposées par les auteurs (au lieu de faire transiter ces valeurs par le leader comme cela est le cas dans le protocole Paxos original). Dans les cas favorables, la combinaison des deux optimisations requièrent seulement 3 étapes de communication.

- **Estimer l'intérêt de l'optimisation risquée**

  Du fait de son caractère risqué, l'optimisation $\mathbf{R_O}$ est l'objet d'une partie de nos travaux. Comme l'optimisation $\mathbf{S_O}$ est sûre, Paxos-MIC intègre l'optimisation $\mathbf{S_O}$ et permet au leader de décider (après évaluation d'un test local) si il déclenche ou pas l'optimisation $\mathbf{R_O}$. Ce protocole permet donc de comparer un comportement de type Paxos + $\mathbf{S_O}$ avec un comportement de type Paxos + $\mathbf{S_O}$+ $\mathbf{R_O}$. Le fait que $\mathbf{R_O}$ soit une optimisation risquée fait que son activation peut conduire à un succès ou à un échec. Dans un premier temps, nous allons montrer que l'ampleur du gain de temps ou de la perte de temps dépend du contexte dans lequel les acteurs ont été déployés. En particulier, le nombre d'acteurs mais aussi leurs positionnements respectifs ont une influence très importante. Une implémentation du protocole Paxos-MIC a été réalisée. Au travers d'exécutions réalisées sur une grille de calcul, le ratio entre gain et perte est calculé pour des contextes types. Ce ratio indique le nombre de succès nécessaires pour pouvoir compenser une perte.

- **Proposition de mécanismes de déclenchement automatique de l'optimisation risquée**

  Nous proposons des critères de déclenchement et évaluons leur qualité, en particulier, leur aptitude à prédire les occurrences de collision. L'ensemble des données expérimentales obtenues sont réutilisées pour tester l'intérêt de ces critères de déclenchement. Cinq critères de base (et certaines de leurs combinaisons) sont analysés en s'appuyant sur un scénario d'utilisation réel. Dans le cas d'un serveur Web sécurisé, nous avons collecté une trace réelle où sont enregistrées les dates d'activation des instances successives de consensus. En nous appuyant sur cette trace ainsi que sur une estimation du risque de collision, nous analysons la qualité de chaque test de déclenchement proposé. Le fait de déclencher l'optimisation $\mathbf{R_O}$ peut être une erreur (conduisant à une perte de temps) ou pas (obtention d'un gain de temps). De même, le fait de ne pas déclencher l'optimisation $\mathbf{R_O}$ peut être une erreur (pas de gain de temps) ou pas (pas de perte de temps). Aucun des tests proposés ne permet de prédire avec

précision l'occurrence d'une collision future. Cependant alors que les tests statiques (toujours activer $\mathbf{R_O}$ ou ne jamais activer $\mathbf{R_O}$) se révèlent incapables de s'adapter à des fluctuations importantes du rythme de déclenchement des consensus, les tests dynamiques que nous proposons permettent pour certains d'obtenir de bons résultats en terme de qualité et en terme d'adaptabilité.

- **Utilisation du consensus comme brique de base**

  Comme nous l'avons mentionné, le consensus est intensivement utilisé comme brique de base pour construire des applications réparties. Nous considérons une application particulière où la séquence de décisions créée permet d'obtenir un coordinateur fiable qui supervise le comportement d'un agent mobile transactionnel. Tandis que l'agent évolue dans un réseau qui peut être un réseau ad-hoc, le coordinateur et ses copies évoluent dans un réseau indépendant. Pour chaque transaction, un agent mobile est créé. L'agent mobile évolue de place en place dans le réseau ad-hoc afin de découvrir un itinéraire où les places visitées pourront satisfaire les requêtes de la transaction. Nous identifions deux services *Disponibilité de la source* et *validation atomique* qui doivent être fournis par le coordinateur extérieur. Leur implémentation repose sur un protocole générant des séquences de décisions persistantes.

- Implémentation et évaluation

  Une part importante du travail a été consacré au développement d'une implémentation efficace du protocole Paxos ainsi qu'à son évaluation. Les expériences ont été réalisées en utilisant la plateforme expérimentale Grid'5000, issue de l'Action de Développement Technologique (ADT) Aladdin pour l'INRIA, avec le support du CNRS, de RENATER, de plusieurs Universités et autres contributeurs (http://www.grid5000.fr).

Dans la suite de ce résumé, nous donnons quelques indications sur le protocole Paxos-MIC, sur les optimisations qu'il intègre ainsi que sur les critères de déclenchement. Nous présentons également quelques résultats obtenus lors de l'évaluation de Paxos-MIC.

I.3 Brève description de Paxos-MIC

Une description détaillée de Paxos-MIC est proposée dans [33, 32]. Nous fournissons ici une brève description des principes généraux et nous soulignons les principales différences avec les autres protocoles de la famille Paxos. Malgré ses spécificités, le comportement de Paxos-MIC correspond soit au comportement de Paxos + $\mathbf{S_O}$, soit au comportement de Paxos + $\mathbf{S_O}$+ $\mathbf{R_O}$.

Deux rôles distincts (*Coordinateur* et *Accepteur*) sont définis et chaque acteur joue un ou deux rôles. Plus précisément, les n acteurs (et de fait, une majorité de noeuds corrects) jouent le rôle d'accepteurs tandis qu'au moins $f + 1$ acteurs (et de fait, au moins un noeud correct) jouent le rôle de coordinateur. Un coordinateur n'est actif que lorsque le service d'élection de leader le désigne comme leader. En pensant alors agir comme un leader unique et incontesté (ce qui n'est pas forcément le cas), le coordinateur tente d'imposer une valeur de décision aux autres acteurs. Dans les protocoles de la famille Paxos, un numéro de tour r

est associé à chaque tentative. Ce numéro est propre au noeud leader N_i qui exécute la tentative: dans notre implémentation, si $1 \leq i \leq n$, la valeur de r choisie par le leader N_i pour identifier son tour courant doit être un multiple de i supérieur à tous les numéros de tour qu'il a pu observer par le passé (les siens et ceux utilisés par d'autres coordinateurs). Dans la version originale du protocole Paxos, un leader tente d'imposer une valeur de décision en exécutant successivement deux phases au cours d'un tour r. Au cours d'une première phase de *Préparation*, le leader s'assure que la valeur qu'il soumettra lors de la seconde phase n'est pas incompatible avec celles éventuellement soumises par d'autres coordinateurs ayant agi comme leader au cours du même consensus mais durant des tours précédents (*i.e.*, des tours dont le numéro est inférieur à r). La phase de préparation implique la diffusion d'un message du leader vers l'ensemble des accepteurs puis la collecte, par le leader, de réponses favorables en provenance d'une majorité d'accepteurs (soit deux étapes de communication). La seconde phase est une phase de *Proposition*. Elle débute une fois que le leader a identifié une valeur qu'il peut soumettre sans risquer de violer les propriétés de sûreté (Accord et Validité). La valeur est diffusée par le leader vers l'ensemble des accepteurs puis le leader attend de collecter une majorité de réponses favorables avant de pouvoir considérer que cette valeur soumise est la valeur de décision (soit à nouveau deux étapes de communication). Un accepteur est une entité passive dont l'état fait référence à la dernière phase de préparation acceptée (numéro de tour r_1) et à la dernière phase de proposition acceptée (numéro de tour r_2 et valeur adoptée). Cet état ne peut évoluer que lors de la réception d'une requête diffusée par un coordinateur et à condition que cette requête soit acceptable. Une requête peut être ignorée par un accepteur si la mise à jour qu'elle entrainerait ne garantit pas i) que la valeur de r_1 croit, ii) que la valeur de r_2 croit ou iii) que $r_1 \leq r_2$ au moment d'une mise à jour de r_2.

Dans sa version originale, le protocole requiert donc 4 étapes de communications entre les acteurs auxquelles viennent s'ajouter 2 étapes "externes" correspondant à la diffusion des valeurs initiales des auteurs vers les coordinateurs et à la diffusion de la valeur de décision du leader vers les auteurs. Le chemin de communication correspond à : auteur $\rightarrow$ coordinateurs (leader) $\rightarrow$ accepteurs $\rightarrow$ leader $\rightarrow$ accepteurs $\rightarrow$ leader $\rightarrow$ auteurs.

L'optimisation $\mathbf{S_O}$ consiste à supprimer une partie du calcul (la phase de préparation) lorsque celle-ci est inutile. Si le leader N_i a réussi à imposer une valeur de décision concernant le consensus $c-1$ durant la tentative r et si, du point de vue de N_i, aucun autre coordinateur N_j ne semble avoir agi avec un numéro de ronde supérieur à r ni durant l'instance de consensus $c-1$ ni durant l'instance c alors N_i peut agir en temps que leader durant le consensus c en utilisant le même numéro de tour r. De fait, un tour r comporte alors une seule phase de préparation qui est suivie par autant de phases de proposition que N_i peut en lancer avant d'être destitué. En conséquence, plusieurs instances de consensus peuvent donc être exécutées durant le même tour. Lorsque le leader élu reste stable (pas de défaillance, interactions avec les autres acteurs suffisamment synchrones), le chemin de communication est de longueur 4 et correspond à : auteur $\rightarrow$ coordinateurs (leader) $\rightarrow$ accepteurs $\rightarrow$ leader $\rightarrow$ auteurs.

L'optimisation $\mathbf{R_O}$ consiste à exécuter de façon anticipée la partie de la phase de proposition correspondant à la diffusion par le leader d'une requête et à sa réception par les accepteurs. L'objectif est de tirer profit du fait que, durant la plupart des instances de consensus, un seul auteur participe au consensus et donc une seule valeur initiale est disponible. Le leader qui ne connait aucune valeur initiale relative au consensus c adopte une valeur par défaut appelée ANY qu'il soumet lors de la phase de proposition. Tout accepteur recevant

cette valeur fictive est alors autorisé à la remplacer par une vraie valeur initiale directement reçue d'un auteur. Lorsque l'activation de $\mathbf{R_O}$ est un succès, le chemin de communication est de longueur 3: auteur $\rightarrow$ accepteurs $\rightarrow$ leader $\rightarrow$ auteurs. Sans $\mathbf{R_O}$, une seule valeur initiale est soumise durant une phase de proposition, et donc tous les accepteurs qui acceptent une valeur durant cette phase adoptent nécessairement la même valeur. Avec $\mathbf{R_O}$, cette propriété n'est plus vérifiée dès lors que deux auteurs fournissent deux valeurs initiales distinctes durant l'instance c. Ceci a deux conséquences majeures. D'une part, même lorsque les conditions sont favorables, la définition d'un quorum est plus contraignante: le leader doit collecter plus d'acceptations [30]. Dans notre implémentation, le taux de retours attendus passe de $\lceil n/2 \rceil$ à $\lceil 3n/4 \rceil$. D'autre part, l'optimisation est un échec lorsque les retours collectés par le leader font référence à plus d'une valeur (occurrence d'une collision). Un recouvrement est alors nécessaire: un nouveau tour est démarré par le coordinateur (via l'exécution d'une phase de préparation) et une nouvelle phase de proposition est lancée mais cette fois, sans activer $\mathbf{R_O}$.

Les auteurs externes qui ne savent pas si l'optimisation est activée ou pas, doivent diffuser systématiquement leur valeur initiale à l'ensemble des coordinateurs et à l'ensemble des accepteurs. Dans le protocole Paxos-MIC, l'activation de $\mathbf{R_O}$ est gérée de la façon suivante. Lorsque l'exécution de l'instance de consensus $c-1$ se termine, un leader qui dispose déjà d'une proposition concernant le prochain consensus c ne déclenche pas $\mathbf{R_O}$ et effectue un consensus en n'utilisant que $\mathbf{S_O}$: nous dirons dans ce cas que les consensus $c-1$ et c s'enchainent. Si au contraire, aucune proposition n'est disponible, le leader est temporairement inactif et il évalue alors un test de déclenchement pour déterminer si $\mathbf{R_O}$ doit être activée ou pas. Si le test est faux (ou plus généralement si il ne devient pas vrai avant qu'une proposition parvienne au leader), le consensus c s'exécute en n'utilisant que $\mathbf{S_O}$: nous dirons dans ce cas que $\mathbf{R_O}$ est non activée (par choix).

Les optimisations $\mathbf{S_O}$ et $\mathbf{R_O}$ ne sont pas des contributions nouvelles. Par contre, le mécanisme d'activation de $\mathbf{R_O}$ à la demande est nouveau. Dans Paxos-MIC, le leader peut choisir en cours d'exécution et pour chaque instance de consensus si l'optimisation risquée $\mathbf{R_O}$ est utilisée ou pas. L'optimisation $\mathbf{S_O}$ a été suggérée dans la description originale du protocole Paxos [37, 38]. Elle est utilisée par Lampson dans [40] (concept de vue) ainsi que par Martin et Alvisi dans [43] (concept de régence). Le protocole FastPaxos (sans espace entre les mots) présenté dans [8] inclut aussi (parmi de nombreuses autres contributions) une implémentation plutôt complexe de l'optimisation $\mathbf{S_O}$. En ce qui concerne l'optimisation $\mathbf{R_O}$, elle a été présenté par Lamport dans un protocole appelé Fast Paxos [39] (avec un espace entre les mots). L'optimisation $\mathbf{R_O}$ est également étudiée dans [25, 30, 18].

Dans [30], les auteurs introduisent la notion de système de quorum raffiné: l'optimisation $\mathbf{R_O}$ est un cas particulier dans ce modèle plus général. Ils considèrent des défaillances byzantines et proposent une solution au problème du consensus qui utilisent les deux optimisations $\mathbf{S_O}$ et $\mathbf{R_O}$ à chaque tour d'une instance de consensus. Dans les scénarios inappropriés où plusieurs auteurs proposent des valeurs initiales différentes, l'utilisation de $\mathbf{R_O}$ peut empêcher une décision durant le tour initiale et retarder le calcul.

Dans [25], les auteurs proposent également une solution qui combine les deux optimisations. Durant chaque tour, le protocole gère simultanément une exécution avec seulement $\mathbf{S_O}$ et une exécution avec à la fois $\mathbf{S_O}$ et $\mathbf{R_O}$. Quelles que soient les circonstances, le temps nécessaire à la prise de décision correspond à la plus rapide de ces deux stratégies menées en parallèle. Cette solution nécessite d'émettre un peu plus de message. De plus,

elle s'avère être adaptée au cas des réseaux large échelle où le coût de communication a un impact important sur le temps nécessaire pour prendre une décision.

Alors que l'optimisation $\mathbf{R_O}$ est systématiquement déclenchée dans [30] et [25], les coordinateurs s'accordent à priori sur le fait qu'un tour utilisera l'optimisation $\mathbf{R_O}$ dans Fast Paxos [39]. Il n'est donc pas question d'adaptation dynamique dans ce protocole. Enfin, dans [18], l'optimisation $\mathbf{R_O}$ est tentée par le leader en charge du tout premier tour et n'est jamais activée lors des tours suivants.

I.4 Critères de déclenchement de $\mathbf{R_O}$

La décision de déclencher ou de ne pas déclencher l'optimisation $\mathbf{R_O}$ pour l'instance de consensus c est prise de manière centralisée par le leader lorsqu'il n'y a pas de valeur de proposition disponible lorsque le consensus $c-1$ se termine. Cette décision conditionne le comportement du consensus c qui sera exécuté. En cas de déclenchement de $\mathbf{R_O}$, les accepteurs se verront octroyer l'autorisation par le leader de renvoyer une valeur au leader sur la base des propositions reçues directement des auteurs. Si un quorum d'accepteurs (égal à $3/4$ des accepteurs) renvoie une valeur unique au leader, cette valeur deviendra la valeur de décision. Dans le cas contraire (absence de quorum d'accepteurs renvoyant une même valeur), il s'agit alors d'une collision et une procédure de recouvrement (coûteuse) sera nécessaire. Le déclenchement de $\mathbf{R_O}$ aura donc été inapproprié. Il apparait donc clairement que la décision de déclencher $\mathbf{R_O}$ constitue un pari sur l'avenir et qu'il est donc crucial de définir ce que nous appellerons désormais un *critère de déclenchement* le plus précis possible afin que le consensus s'exécute à chaque fois en un temps le plus court possible.

I.4.1 Critères dynamiques

Nous définissons des critères de déclenchement génériques dépendant du comportement récent du protocole de consensus. Ces critères sont indépendants de l'application ou du protocole utilisant le consensus et ne nécessitent tout au plus qu'une mémorisation du comportement des instances précédentes du consensus. Deux critères simples, pouvant être évalués très rapidement par le leader, sont définis. Le premier de ces critères, dénommé *Time* déclenche $\mathbf{R_O}$ s'il n'y a pas de valeur de proposition disponible avant un délai d'au moins Δ ms après la fin du dernier consensus. On profite ici d'une période d'accalmie (plus ou moins relative en fonction de la valeur attribuée à Δ) pour anticiper sur l'absence de collision lors du prochain consensus. Le second critère, dénommé *Result* déclenche $\mathbf{R_O}$ sauf s'il y a eu au moins une collision durant les k derniers consensus exécutés. La démarche est ici différente de celle adoptée pour le critère *Time* dans la mesure où on ne profite pas nécessairement d'une "accalmie" mais d'une configuration débouchant rarement sur des collisions (cela peut être lié par exemple à la localisation d'un quorum d'accepteurs favorisant l'absence de collisions même en présence de différentes valeurs de proposition). Un troisième critère, dénommé *Random* déclenche $\mathbf{R_O}$ en utilisant une probabilité fixe de déclenchement p et n'utilise donc pas une quelconque connaissance du passé pour décider s'il faut déclencher $\mathbf{R_O}$ ou pas.

Réglages des critères

Les deux critères *Time* et *Result* peuvent être réglés pour déclencher $\mathbf{R_O}$ de façon plus ou moins fréquentes. Ainsi, si la procédure de recouvrement à exécuter en cas de collision est relativement peu coûteuse et si le gain apporté par l'optimisation $\mathbf{R_O}$ en cas de déclenchement approprié (pas de collision) est élevé, on peut être plus agressif en favorisant le déclenchement de $\mathbf{R_O}$. Pour cela, fixer à Δ à une valeur très faible (critère *Time*) ou fixer k à 1 pour le critère *Result* peuvent constituer des choix intéressants. Inversement, si la procédure de recouvrement est coûteuse et si le gain apporté par un déclenchement approprié de $\mathbf{R_O}$ est faible, on peut adopter une attitude plus prudente en fixant Δ à une valeur élevée (critère *Time*) ou k à une valeur élevée (critère *Result*).

I.4.2 Critères statiques

Nous définissons également deux critères statiques dont l'évaluation donne toujours les mêmes résultats: *Never* et *Always*. Lorsqu'il est évalué, le critère *Never* ne déclenche pas $\mathbf{R_O}$. Inversement, quand il est évalué, le critère *Always* déclenche $\mathbf{R_O}$.

I.5 Conditions d'expérimentation

Le protocole Paxos-MIC, dont le pseudo-code est disponible dans [33] a été mis en œuvre en langage Java. Nos expérimentations ont été conduites sur la plateforme expérimentale Grid'5000 qui comporte plus de vingt clusters répartis sur dix sites en France. Dans un cluster, les nœuds de calcul sont homogènes (même processeur) mais les processeurs peuvent différer d'un cluster à un autre. Tous les nœuds de calcul exécutent le système Debian. Nous utilisons la machine virtuelle OpenJDK6 (compilateur JIT désactivé) pour réaliser nos mesures sur les nœuds de la grille.

RTT (en ms)	Lille	Toulouse	Orsay	Rennes
Lille	-	19.8	4.75	10.58
Toulouse	19.8	-	16.51	22.3
Orsay	4.75	16.51	-	9.18
Rennes	10.58	22.3	9.18	-

Table I.1: RTT inter-sites

Deux configurations réseau ont été prises en compte pour les mesures de Paxos-MIC. La première configuration est mono-site (Rennes) au sein de laquelle, le RTT (Round-Trip Time) entre deux nœuds est peu élevé (moins de 0.140 ms). La seconde configuration est multi-sites. Elle interconnecte plusieurs sites Grid'5000 (Orsay, Lille, Toulouse, Grenoble et Rennes ont été retenus). Dans cette configuration, le RTT entre deux nœuds appartenant à deux sites différents peut aller jusqu'à 20 ms (22.3 ms entre Rennes et Toulouse).

Nos expérimentations mettent l'accent sur deux facteurs qui ont un impact direct sur les performances du protocole Paxos-MIC: la localisation géographique des différents acteurs et la taille du groupe d'acteurs (nombre d'accepteurs et de coordinateurs). D'autres

facteurs, comme l'occurence de défaillances et le délai moyen entre consensus ont également été étudiés.

I.5.1 Quatre contextes de référence

Sur la base des observations précédentes, nous identifions quatre contextes différents. Dans deux d'entre eux (C_{L5} et C_{W5}), le groupe d'acteurs est constitué de 3 coordinateurs et 5 accepteurs. Dans les deux autres, le groupe d'acteurs est constitué de 6 coordinateurs et 11 accepteurs. Dans deux d'entre eux (C_{L5} et C_{L11}), les acteurs sont sur le même site (Rennes). Dans les deux autres, les auteurs-apprenants externes sont sur le même site, les coordinateurs et une minorité d'accepteurs sont sur un second site et une majorité d'accepteurs est sur un troisième site. Le tableau I.2 indique les durées de consensus qui ont été observées pendant les expérimentations: D_{succ} ($\mathbf{R_O}$ est déclenchée et aucune collision ne survient), D_{norm} ($\mathbf{R_O}$ n'est pas déclenchée), et D_{fail} ($\mathbf{R_O}$ est déclenchée et une collision se produit). Dans les quatre contextes, l'optimisation $\mathbf{R_O}$ peut présenter un intérêt: $D_{succ} < D_{norm} < D_{fail}$. Le ratio est défini comme étant (D_{fail} - D_{norm} / D_{norm} - D_{succ}). Il indique le nombre de succès nécessaires pour compenser un échec.

	D_{succ}	D_{norm}	D_{fail}	ratio (échec/succès)
Contexte C_{L5}	1.31ms	1.92ms	3.15ms	2.0
Contexte C_{L11}	1.78ms	2.12ms	5.18ms	8.8
Contexte C_{W5}	18.68ms	25.88ms	42.20ms	2.3
Contexte C_{W11}	21.59ms	26.01ms	42.82ms	3.8

Table I.2: Durée moyenne du consensus

I.6 Déclenchement de $\mathbf{R_O}$: analyse dans le cadre d'une application WEB

Etant donné que l'un des principaux intérêts de Paxos-MIC réside dans sa capacité à déclencher $\mathbf{R_O}$ lorsque cela est approprié (et à ne pas le déclencher quand cela n'est pas approprié), nous mettons désormais l'accent sur l'analyse des critères de déclenchement de $\mathbf{R_O}$. Dans cette section, nous cherchons à évaluer différents critères de déclenchement dans le contexte d'une application réelle nécessitant pour sa progression l'utilisation d'un algorithme de consensus. L'application consiste en une architecture WEB répliquée dans laquelle les requêtes HTTP doivent être ordonnées, grâce au protocole Paxos-MIC avant d'être envoyées aux serveurs WEB. Ici, à la différence des mesures réalisées à la section I.6.4, nous n'exécutons pas le code du protocole Paxos-MIC. Nous utilisons un log de requêtes HTTP qui ont été collectées auprès du serveur WEB d'une école d'ingénieurs durant une période de 16 jours. En analysant le log, sur la base des temps d'arrivée des requêtes qui y sont enregistrées et en tenant naturellement compte des différentes durées D_{norm}, D_{succ}et D_{fail}, nous calculons:

- le nombre de déclenchements de $\mathbf{R_O}$,

- les nombres de succès et d'échecs dans les déclenchements de $\mathbf{R_O}$,
- les nombres de succès et d'échecs dans les non déclenchements de $\mathbf{R_O}$.

I.6.1 Critères de déclenchement

L'avantage de cette analyse de log par rapport à une exécution réelle du protocole pour ordonner les requêtes du log est que cela nous permet de tester des critères dits *irréalistes* nous servant ainsi de points de comparaison avec les critères dits *réalistes* définis à la section I.4. Les critères irréalistes peuvent être évalués en analysant le log mais ne peuvent pas être mis en œuvre car ils nécessitent une connaissance du futur. Nous en définissons deux: *Optimal* et *Worst*.

Le critère *Optimal* déclenche $\mathbf{R_O}$ seulement s'il n'y aura pas de collision dans le futur et donne ainsi la meilleure performance possible (dans le cas de notre analyse). A la différence du critère *Optimal*, le critère *Worst* déclenche $\mathbf{R_O}$ seulement s'il y aura une collision et ne déclenche pas $\mathbf{R_O}$ sinon. Nous utilisons ces deux critères pour définir une échelle permettant d'évaluer comparativement les performances des cinq autres critères (*Always*, *Never*, *Random*, *Result* et *Time*). Si t est la durée moyenne du consensus pour une activation de $\mathbf{R_O}$ selon un critère donné C, t_{worst}, la durée moyenne selon le critère *Worst* et $t_{optimal}$, la durée moyenne selon le critère *Optimal*, le gain pour le critère C est $(t_{worst} - t)/(t_{worst} - t_{optimal})$.

Par ailleurs, pour les critères réalistes définis à la section I.4, nous avons fait les choix suivants. Pour le critère *Time*, Δ est fixé à 10ms: si une requête arrive dans un délai de 10ms après la fin du consensus précédent, $\mathbf{R_O}$ ne sera pas déclenchée. Concernant le critère *Result*, k est fixé à 2. Il faut donc 2 consensus consécutifs sans collision pour que le troisième consensus soit exécuté avec l'optimisation $\mathbf{R_O}$. Enfin, dans le cas du critère *Random*, p est fixé à 0.8.

I.6.2 Logs

Pendant la période des 16 jours de collecte du log, un total de 573753 requêtes ont été adressées au serveur Web. La fréquence des requêtes n'est pas très élevée (environ 25 requêtes par minute en moyenne) mais cette fréquence n'est pas uniforme sur la totalité du log: les périodes "nuit" ont considérablement moins de requêtes que les périodes "jour". Nous avons trouvé dans le log une période de 10 heures qui est six fois plus dense que le log entier. Comme cela n'est cependant pas suffisant pour représenter un serveur fortement chargé, nous avons décidé de calculer un nouveau log où l'intervalle de temps entre deux requêtes consécutives du log original est divisée par 60. Ainsi, la fréquence moyenne des requêtes est d'environ 1500 requêtes par minute dans ce nouveau log.

Nous disposons donc de deux logs: le log original et le log compressé sur lesquels nous pouvons tester nos critères de déclenchement.

I.6.3 Contextes

Quatre contextes précédement définis sont considérés et pour chacun de ces contextes, nous évaluons la qualité des différents critères de déclenchement de $\mathbf{R_O}$. Nous estimons le risque de collision de la manière suivante: soient p_i et p_j deux requêtes qui apparaissent dans le

log. Si $\mathbf{R_O}$ est activée lors du consensus c destiné à ordonner p_i, nous considérons qu'il y a un risque de collision si p_j est proposé avant que le consensus c termine.

I.6.4 Résultats et analyses

Le comportement du leader conduit à classer une instance c de consensus dans l'une des trois classes suivantes:

1. *Consensus immédiat*: le leader n'évalue pas le critère de déclenchement parce qu'un valeur initiale était déjà disponible à la fin du consensus $c-1$. $\mathbf{R_O}$ n'est pas déclenchée et la durée du consensus c est égale à D_{norm}.

2. *Consensus Type 1*: le leader évalue le critère de déclenchement à faux. $\mathbf{R_O}$ n'est pas déclenchée et la durée du consensus c est égale à D_{norm}. La prédiction est soit juste soit erronée. Bien sûr, durant le calcul, le leader ne peut pas savoir (même a posteriori) si une erreur a été commise ou pas. Cependant, au cours de notre analyse, nous considérons le risque de collision afin de distinguer les prédictions qui sont justes de celles qui sont erronées (un gain était possible).

3. *Consensus Type 2*: le leader évalue le critère de déclenchement à vrai. $\mathbf{R_O}$ est déclenchée. Si aucune collision ne se produit, le déclenchement est un succès et la durée du consensus c est égale à D_{succ}. Dans le cas contraire, le déclenchement est un échec et la durée du consensus est égale à D_{fail}.

Lors de notre analyse, chaque instance du consensus appartient à l'une des cinq classes suivantes: immédiat, succès type 1, erreur type 1, succès type 2 ou erreur type 2. Rappelons que pour évaluer le risque d'une collision, nous adoptons la règle suivante. Une collision se produit durant l'instance c du consensus si $\mathbf{R_O}$ est déclenchée durant ce consensus et au moins une autre propostion est générée avant la fin du consensus.

Nous évaluons les critères de déclenchement en mesurant le gain obtenu sur la durée moyenne du consensus pour les quatre contextes et avec les deux logs. Le tableau I.3 donne tous les gains mesurés.

	Log original					Log compressé				
	Never	*Always*	*Random*	*Time*	*Result*	*Never*	*Always*	*Random*	*Time*	*Result*
C_{L5}	4.8%	95.1%	77%	89.7%	93.8%	25.2%	70.7%	60.8%	52.3%	69.7%
C_{L11}	19.9%	80%	67.9%	77.3%	80.1%	62.9 %	31.8%	37%	62.2%	35.1%
C_{W5}	18.9%	80.4%	68%	79%	78.8%	53 %	42.8%	44.7%	44.3%	44%
C_{W11}	29.9%	69.5%	61.4%	68.8%	70.1%	67.1 %	29.9%	36.9%	33.1%	33.5%

Table I.3: Gains sur une échelle *Worst* (0%) - *Optimal* (100%)

Concernant le log original, pour tous les contextes, les gains sont élevés avec les critères *Random*, *Always*, *Time* et *Result*. La comparaison de ces gains avec le gain obtenu avec le critère *Never* démontre sans ambiguité l'intérêt de $\mathbf{R_O}$. Lorsque l'on considère le log compressé, le gain demeure élevé pour le contexte C_{L5} avec les critères *Always*, *Random*, *Time* et *Result*. Il devient cependant plus faible qu'avec le critère *Never* pour les trois autres contextes C_{L11}, C_{W5} et C_{W11}.

Nous avons également mesuré le taux de succès et d'erreurs (type 1 et 2) ainsi que le taux de consensus immédiats pour chaque critère de déclenchement dans chaque contexte et ceci pour les deux logs.

Quel critère ?

Jusqu'à présent, nous n'avons pas discuté du critère *Random*. Ce critère a été introduit pour vérifier si les résultats obtenus avec les critères *Time* et *Result* étaient davantage dus à leur acuité (ou précision ?) plutôt qu'au nombre d'activations de $\mathbf{R_O}$. Pour le log original, le gain avec le critère random est toujours inférieur à celui obtenu avec les critères *Time* et *Result*. Pour le log compressé, le gain avec se situe entre celui avec *Result* et *Time* pour les contextes C_{L5} et C_{L11}. Comme expliqué précédemment, les gains pour C_{W11} et C_{W5} ne sont pas significatifs. Le critère *Time* nécessite de régler la durée du paramètre Δ pour s'adapter aux caractéristiques du contexte. Clairement, 10ms est un choix approprié pour C_{L5} et C_{L11} mais pas pour C_{W5} et C_{W11}. Le critère *Result* ne requiert pas de réglage et permet d'obtenir un gain relativement important quel que soit le contexte. Des combinaisons des critères (conjonction ou disjonction) ont été évaluées; elles n'améliorent pas significativement les résultats et requièrent de sélectionner a priori s'il est préférable d'augmenter ou de diminuer le nombre de déclenchements de $\mathbf{R_O}$.

I.6.5 Evaluation du risque

Lors de l'analyse, nous avons utilisé une approche assez pessimiste concernant l'évaluation du risque de collision: si une proposition p_j est générée (correspondant à une arrivée de requête HTTP) avant la fin d'une instance c d'un consensus (pour lequel une proposition p_i avait été initialement diffusée), nous considérons qu'une collision survient. Ce choix conduit clairement à une surestimation du risque de collision. En effet, si une proposition p_j est générée très peu de temps avant la fin d'une instance c du consensus, il est très fortement probable qu'elle n'induira pas de collision pour cette instance c (un quorum d'accepteurs ayant eu suffisamment de temps pour faire de la proposition initiale p_i une valeur décidée avant la diffusion de p_j). Nous avons donc souhaité vérifier si notre approche (pessimiste) faussait les résultats et notamment les performances relatives des différents critères de déclenchement. Pour cela, nous avons utilisé deux nouvelles versions du risque de collision. Dans une première version, nous avons considéré qu'une collision ne pouvait se produire que si une proposition p_j était générée pendant la première moitié de l'exécution du consensus c. Dans une seconde version, nous avons considéré que la génération d'une proposition p_j conduisait à une collision avec une probabilité décroissante au fil de l'exécution du consensus c (égale à 1 au début de l'exécution et à 0 vers la fin de l'exécution).

Avec ces deux nouvelles versions du risque de collision, nous n'avons observé que de très légères variations des résultats par rapport à ce que nous avons obtenu avec la version jugée pessimiste de notre évaluation du risque de collision.

I.7 Conclusion

Dans ce résumé, nous avons vu que le protocole Paxos-MIC combine l'utilisation de deux optimisations. Il permet de résoudre plusieurs instances de consensus et de garantir la persistance des valeurs décidées. Le protocole est adaptatif dans la mesure où il vise à obtenir le meilleur gain de performances en fonction du contexte. Entre deux instances consécutives du consensus, le leader détermine si l'optimisation $\mathbf{R_O}$ doit être déclenchée ou pas.

Le document complet comporte d'autres contributions (notamment une étude sur l'utilisation du consensus dans le cas d'agents mobiles transactionnels).

Parmi nos perspectives de recherche, nous pouvons mentionner: la prise en compte des comportements arbitraires, l'anonymat des participants, une étude pratique du risque de collision, la proposition d'un mécanisme générique de prise de décision, et enfin l'étude des mécanismes d'oublis du préfixe de la séquence de décisions.

VU:

Le Directeur de Thèse

Michel HURFIN

VU:

Le Responsable de l'École Doctorale

Jean-Marie LION

VU pour autorisation de soutenance

Rennes, le

Le Président de l'Université de Rennes 1

Guy CATHELINEAU

VU après soutenance pour autorisation de publication:

Le Président du jury,

César VIHO