Revisiting the Paxos Algorithm

by

Roberto De Prisco

Laurea in Computer Science (1991)
University of Salerno, Italy

Submitted to the Department of
Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Science in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

June 3, 1997

Revised August 1998

Author ....
Department of
Electrical Engineering and Computer Science
June 3, 1997

Certified by ....
Prof. Nancy Lynch
NEC Professor of Software Science and Engineering
Thesis Supervisor

Accepted by ....
Prof. Arthur C. Smith
Chairman, Department Committee on Graduate Theses

Revisiting the Paxos Algorithm

by

Roberto De Prisco

Submitted to the Department of
Electrical Engineering and Computer Science
on June 3, 1997, in partial fulfillment of the
requirements for the degree of
Master of Science in Electrical Engineering and Computer Science

Abstract

The PAXOS algorithm is an efficient and highly fault-tolerant algorithm, devised by Lamport, for reaching consensus in a distributed system. Although it appears to be practical, it seems to be not widely known or understood. This thesis contains a new presentation of the PAXOS algorithm, based on a formal decomposition into several interacting components. It also contains a correctness proof and a time performance and fault-tolerance analysis.

The presentation is built upon a general timed automaton (GTA) model. The correctness proof uses automaton composition and invariant assertion methods. The time performance and fault-tolerance analysis is conditional on the stabilization of the underlying physical system behavior starting from some point in an execution. In order to formalize this stabilization, a special type of GTA called a Clock GTA is defined.

Thesis Supervisor: Prof. Nancy Lynch
Title: NEC Professor of Software Science and Engineering

Acknowledgments

I would like to thank my advisor Nancy Lynch. Her constant support throughout this work, her help for my understanding of the subject and the writing of the thesis, her thorough review of various revisions of the manuscript, substantially improved the overall quality of the thesis. I also thank Nancy for her patience in explaining me many times the things that I did not understand readily. In particular I am grateful to her for teaching me the difference between "that" and "which", that (or which?) is something I still have to understand!

I would also like to thank Butler Lampson for useful discussions and suggestions. Butler also has written the 6.826 Principles of Computer System handout that provides a description of PAXOS; that handout provided the basis for the work in this thesis.

I would like to thank Alan Fekete, Victor Luchangco, Alex Shvartsman and Mandana Vaziri for reading parts of the thesis and Atul Adya and Arvind Parthasarathi for useful discussions.

Contents

1 Introduction **7**

2 Models **16**

2.1 Overview 16

2.2 The basic I/O automata model 18

2.3 The MMT automaton model. 19

2.4 The GT automaton model 21

2.5 The Clock GT automaton model 23

2.6 Composition of automata 29

2.6.1 Composition of BIOA. 29

2.6.2 Composition of MMTA. 32

2.6.3 Composition of GTA. 33

2.6.4 Composition of Clock GTA. 35

2.7 Bibliographic notes 35

3 The distributed setting **36**

3.1 Processes 37

3.2 Channels 38

3.3 Distributed systems 41

4 The consensus problem **44**

4.1 Overview 44

4.2 Formal definition 47

4.3 Bibliographic notes 48

5 Failure detector and leader elector **50**

5.1 A failure detector 50

5.2 A leader elector 56

5.3 Bibliographic notes 59

6 The PAXOS algorithm **61**

6.1 Overview 61

6.2 Automaton BASICPAXOS 64

6.2.1 Overview 64

6.2.2 The code 70

6.2.3 Partial Correctness 80

6.2.4 Analysis of S_{BPX} 91

6.3 Automaton S_{PAX} 97

6.4 Correctness and analysis of S_{PAX} 100

6.5 Messages 104

6.6 Concluding remarks 105

7 The MULTIPAXOS algorithm **107**

7.1 Overview 107

7.2 Automaton MULTIBASICPAXOS. 108

7.3 Automaton MULTISTARTERALG 118

7.4 Correctness and analysis 119

7.5 Concluding remarks 119

8 Application to data replication **122**

8.1 Overview 122

8.2 Sequential consistency 123

8.3 Using MULTIPAXOS 124

8.3.1 The code 126

8.3.2 Correctness and analysis 130

8.4 Concluding remarks 132

9 Conclusions **134**

A Notation **138**

List of Figures

2-1 An I/O automaton. 21

3-1 Automaton CHANNEL$_{i,j}$ 42

3-2 The communication system S_{CHA} 45

5-1 Automaton DETECTOR for process i 54

5-2 The system S_{DET} 55

5-3 Automaton LEADERELECTOR for process i 60

5-4 The system S_{LEA} 61

6-1 PAXOS: process i. 66

6-2 BASICPAXOS: Messages. 66

6-3 Exchange of messages 69

6-4 Choosing the values of rounds. 72

6-5 Automaton BPLEADER for process i (part 1) 74

6-6 Automaton BPLEADER for process i (part 2) 75

6-7 Automaton BPAGENT for process i 76

6-8 Automaton BPSUCCESS for process i (part 1) 77

6-9 Automaton BPSUCCESS for process i (part 2) 78

6-10 History variables. 84

6-11 Automaton STARTERALG for process i 102

7-1 Automaton BMPLEADER for process i (part 1) 115

7-2 Automaton BMPLEADER for process i (part 2) 116

7-3 Automaton BMPAGENT for process i 117

7-4 Automaton BMPSUCCESS for process i (part 1) 118
7-5 Automaton BMPSUCCESS for process i (part 2) 119
7-6 Automaton MULTISTARTERALG for process i (part 1) 123
7-7 Automaton MULTISTARTERALG for process i (part 2) 124

8-1 Automaton DATAREPLICATION for process i (part 1) 130
8-2 Automaton DATAREPLICATION for process i (part 2) 131
8-3 Data replication 136

Chapter 1

Introduction

Reaching consensus is a fundamental problem in distributed systems. Given a distributed system in which each process[1] starts with an initial value, to solve a consensus problem means to give a distributed algorithm that enables each process to eventually output a value of the same type as the input values, in such a way that three conditions, called *agreement*, *validity* and *termination*, hold. There are different definitions of the problem depending on what these conditions require. The agreement condition states requirements about the way processes need to agree (e.g., "no two different outputs occur"). The validity condition states requirements about the relation between the input and the output values (e.g., "any output value must belong to the set of initial values"). The termination condition states requirements about the termination of an algorithm that solves the problem (e.g., "each non-faulty process eventually outputs a value"). Distributed consensus has been extensively studied; a good survey of early results is provided in [17]. We refer the reader to [35] for a more up-to-date treatment of consensus problems.

[1]We remark that the words "process" and "processor" are often used as synonyms. The word "processor" is more appropriate when referring to a physical component of a distributed system. A physical processor is often viewed as consisting of several logical components, called "processes". Processes are composed to describe larger logical components, and the resulting composition is also called a process. Thus the whole physical processor can be identified with the composition of all its logical components. Whence the word "process" can also be used to indicate the physical processor. In this thesis we use the word "process" to mean either a physical processor or a logical component of it. The distinction either is unimportant or should be clear from the context.

Consensus problems arise in many practical situations, such as, for example, distributed data replication, distributed databases and flight control systems. Data replication is used in practice to provide high availability: having more than one copy of the data allows easier access to the data, i.e., the nearest copy of the data can be used. However, consistency among the copies must be maintained. A consensus algorithm can be used to maintain consistency. A practical example of the use of data replication is an airline reservation system. The data consists of the current booking information for the flights and it can be replicated at agencies spread over the world. The current booking information can be accessed at any of the replicas. Reservations or cancellations must be agreed upon by all the copies.

In a distributed database, the consensus problem arises when a collection of processes participating in the processing of a distributed transaction has to agree on whether to commit or abort the transaction, that is, make the changes due to the transaction permanent or discard the changes. A common decision must be taken to avoid inconsistencies. A practical example of the use of distributed transactions is a banking system. Transactions can be done at any bank location or ATM machine, and the commitment or abortion of each transaction must be agreed upon by all the bank locations or ATM machines involved.

In a flight control system, the consensus problem arises when the flight surface and airplane control systems have to agree on whether to continue or abort a landing in progress or when the control systems of two approaching airplanes need to modify the air routes to avoid collision.

Various theoretical models of distributed systems have been considered. A general classification of models is based on the kind of communications allowed between processes of the distributed system. There are two ways by which processes communicate: by passing messages over communication channels or using a shared memory. In this thesis we focus on message-passing models.

A wide variety of message-passing models can be used to represent distributed systems. They can be classified by the network topology, the synchrony of the system and the failures allowed. The network topology describes which processes can send messages directly to which other processes and it is usually represented by a graph in which nodes represent processes and edges represent direct communication channels. Often one assumes that a process knows the entire network; sometimes one assumes that a process has only a local knowledge of the network (e.g., each process knows only the processes for which it has a direct communication channel).

About synchrony, several model variations, ranging from the completely asynchronous setting to the completely synchronous one, can be considered. A completely asynchronous model is one with no concept of real time. It is assumed that messages are eventually delivered and processes eventually respond, but it may take arbitrarily long. In partially synchronous systems some timing assumptions are made. For example, upper bounds on the time needed for a process to respond and for a message to be delivered hold. These upper bounds are known by the processes and processes have some form of real-time clock to take advantage of the time bounds. In completely synchronous systems, the computation proceeds in rounds in which steps are taken by all the processes.

Failures may concern both communication channels and processes. In partially synchronous models, messages are supposed to be delivered and processes are expected to act within some time bounds; a *timing failure* is a violation of these time bounds. Communication failures can result in loss of messages. Duplication and reordering of messages may be considered failures, too. The weakest assumption made about process failures is that a faulty process has an unrestricted behavior. Such a failure is called a *Byzantine failure*. More restrictive models permit only *omission failures*, in which a faulty process fails to send some messages. The most restrictive models allow only *stopping failures*, in which a failed process simply stops and takes no further actions. Some models assume that failed processes can be restarted. Often processes have some form of stable storage that is not affected by a stopping failure; a stopped process is restarted with its stable storage in the same state as before the failure and with every other part of its state restored to some initial values.

In the absence of failures, distributed consensus problems are easy to solve: it is enough to exchange information about the initial values of the processes and use a common decision rule for the output in order to satisfy both agreement and validity. Failures complicate the matter, so that distributed consensus can even be impossible to achieve. The difficulties depend upon the distributed system model considered and the exact definition of the problem (i.e., the agreement, validity and termination conditions).

Real distributed systems are often partially synchronous systems subject to process, channel and timing failures and process recoveries. Today's distributed systems occupy larger and larger physical areas; the larger the physical area spanned by the distributed system, the harder it is to provide synchrony. Physical components are subject to failures. When a failure occurs, it is likely that, some time later, the problem is fixed, restoring the failed component to normal operation. Moreover, though timely responses can usually be provided in real distributed systems, the possibility of process and channel failures makes it impossible to guarantee that timing assumptions are always satisfied. Thus real distributed systems suffer timing failures. Any practical consensus algorithm needs to consider all the above practical issues. Moreover, the basic safety properties must not be affected by the occurrence of failures. Also, the performance of the algorithm should be good when there are no failures.

PAXOS is an algorithm devised by Lamport [29] that solves the consensus problem. The model considered is a partially synchronous distributed system where each process has a direct communication channel with each other process. The failures allowed are timing failures, loss, duplication and reordering of messages and process stopping failures. Process recoveries are considered; some stable storage is needed. PAXOS is guaranteed to work safely, that is, to satisfy agreement and validity, regardless of process, channel and timing failures and process recoveries. When the distributed system stabilizes, meaning that there are no failures nor process recoveries and a majority of the processes are not stopped, for a sufficiently long time, termination is achieved; the performance of the algorithm when the system stabilizes is good. In [29] there is also presented a variation of PAXOS that considers multiple concurrent runs of PAXOS when consensus has to be reached on a sequence of values. We call this variation the

MULTIPAXOS algorithm[2].

The basic idea of the PAXOS algorithm is to have processes propose values until one of the value is accepted by a majority of the processes: that value is the final output value. Any process may propose a value by initiating a *round* for that value. The process initiating a round is the *leader* of that round. Rounds are guaranteed to satisfy agreement and validity. A successful round, that is, a round in which a value is accepted by a majority of the processes, results in the termination of the algorithm. However a successful round is guaranteed to be conducted only when the distributed system stabilizes. Basically PAXOS keeps starting rounds while the system is not stable, but when the system stabilizes, a successful round is conducted. Though failures may force the algorithm to repeatedly start new rounds, a single round is not costly: it uses $O(n)$ messages, where n is the number of processes, and constant time. Thus, PAXOS has good fault-tolerance properties and when the system is stable combines those fault-tolerance properties with the performance of an efficient algorithm, so that it can be useful in practice.

In the original paper [29], the PAXOS algorithm is described as the result of discoveries of archaeological studies of an ancient Greek civilization. That paper contains a sketch of a proof of correctness and a discussion of the performance analysis. The style used for the description of the algorithm often diverts the reader's attention. Because of this, we found the paper hard to understand and we suspect that others did as well. Indeed the PAXOS algorithm, even though it appears to be a practical and elegant algorithm, seems not widely known or understood, either by distributed systems researchers or distributed computing theory researchers.

This thesis contains a new, detailed presentation of the PAXOS algorithm, based on a formal decomposition into several interacting components. It also contains a correctness proof and a time performance and fault-tolerance analysis. The MULTIPAXOS algorithm is also described together with an application to data replication.

[2]PAXOS is the name of the ancient civilization studied in [29]. The actual algorithm is called the "single-decree synod" protocol and its variation for multiple consensus is called the "multi-decree parliament" protocol. We take the liberty of using the name PAXOS for the single-decree synod protocol and the name MULTIPAXOS for the multi-decree parliament protocol.

The formal framework used for the presentation is provided by Input/Output automata models. Input/Output automata are simple state machines with transitions labelled with actions. They are suitable for describing asynchronous and partially synchronous distributed systems. The basic I/O automaton model, introduced by Lynch and Tuttle [37], is suitable for modelling asynchronous distributed systems. For our purposes, we will use the general timed automaton (GTA) model, introduced by Lynch and Vandraager [38, 39, 40][3], which has formal mechanisms to represent the passage of time and is suitable for modelling partially synchronous distributed systems.

The correctness proof uses automaton composition and invariant assertion methods. Composition is useful for representing a system using separate components. This split representation is helpful in carrying out the proofs. We provide a modular presentation of the PAXOS algorithm, obtained by decomposing it into several components. Each one of these components copes with a specific aspect of the problem. In particular there is a "failure detector" module that detects process failures and recoveries. There is a "leader elector" module that copes with the problem of electing a leader; processes elected leader by this module, start new rounds for the PAXOS algorithm. The PAXOS algorithm is then split into a basic part that ensures agreement and validity and an additional part that ensures termination when the system stabilizes; the basic part of the algorithm, for the sake of clarity of presentation, is further subdivided into three components. The correctness of each piece is proved by means of invariants, i.e., properties of system states that are always true in an execution. The key invariants we use in our proof are the same as in [31, 32].

The time performance and fault-tolerance analysis is conditional on the stabilization of the system behavior starting from some point in an execution. While it is easy to formalize process and channel failures, dealing formally with timing failures is harder. To cope with this problem, this thesis introduces a special type of GTA called a Clock GTA. The Clock GTA is a GTA augmented with a simple way of for-

[3]The term "general timed automaton" was not used in these papers; it was introduced in the book by Lynch [35].

malizing timing failures. Using the Clock GTA we provide a technique for practical time performance analysis based on the stabilization of the physical system.

A detailed description of the MULTIPAXOS protocol is also provided. As an example of an application, the use of MULTIPAXOS to implement a data replication algorithm is presented. With MULTIPAXOS the high availability of the replicated data is combined with high fault tolerance. This is not trivial, since having replicated copies implies that consistency has to be guaranteed and this may result in low fault tolerance.

Independent work related to PAXOS has been carried out. The algorithms in [11, 34] have similar ideas. The algorithm of Dwork, Lynch and Stockmeyer [11] also uses rounds conducted by a leader, but the rounds are conducted sequentially, whereas in PAXOS a leader can start a round at any time and multiple simultaneous leaders are allowed. The strategy used in each round by the algorithm of [11] is somewhat different from the one used by PAXOS. Moreover the distributed model of [11] does not consider process recoveries. The time analysis provided in [11] is conditional on a "global stabilization time" after which process response times and message delivery times satisfy the time assumptions. This is similar to our analysis. (A similar time analysis, applied to a different problem, can be found in [16].)

MULTIPAXOS can be easily used to implement a data replication algorithm. In [34] a data replication algorithm is provided. It incorporates ideas similar to the ones used in PAXOS.

PAXOS bears some similarities with the standard three-phase commit protocol: both require, in each round, an exchange of 5 messages. However the standard commit protocol requires a reliable leader elector while PAXOS does not. Moreover PAXOS sends information on the value to agree on, only in the third message of a round, while the commit protocol sends it in the first message; because of this, MULTIPAXOS can exchange the first two messages only once for many instances and use only the exchange of the last three messages for each individual consensus problem while such a strategy cannot be used with the three-phase commit protocol.

In the class notes of the graduate level Principles of Computer Systems course [31] taught at MIT, a description of PAXOS is provided using a specification language called

SPEC. The presentation in [31] contains the description of how a round of PAXOS is conducted. The leader election problem is not considered. Timing issues are not addressed; for example, the problem of starting new rounds is not considered. A proof of correctness, written also in SPEC, is outlined. Our presentation differs from that of [31] in the following aspects: it is based on I/O automata models rather than on a programming language; it provides all the details of the algorithm; it provides a modular description of the algorithm, including auxiliary parts such as a failure detector module and a leader elector module; along with the proof of correctness, it provides a performance and fault-tolerance analysis. In [32] Lampson provides an overview of the PAXOS algorithm together with the key points for proving the correctness of the algorithm.

In [43] the clock synchronization problem has been studied; the solution provided there introduces a new type of GTA, called the mixed automaton model. The mixed automaton is similar to our Clock automaton with respect to the fact that both try to formally handle the local clocks of processes. However while the mixed automaton model is used to describe synchronization of the local clocks, the Clock GTA automaton is used to model good timing behavior and thus does not need to cope with synchronization.

Summary of contributions. This thesis provides a new, detailed and modular description of the PAXOS algorithm, a correctness proof and a time performance analysis. The MULTIPAXOS algorithm is described and an application to data replication is provided. This thesis also introduces a special type of GTA model, called the Clock GTA model, and a technique for practical time performance analysis when the system stabilizes.

Organization. This thesis is organized as follows. In Chapter 2 we provide a description of the I/O automata models and in particular we introduce the Clock GTA model. In Chapter 3 we discuss the distributed setting we consider. Chapter 4 gives a formal definition of the consensus problem we consider. Chapter 5 is devoted to the design of a simple failure detector and a simple leader elector which will be used to give an implementation of PAXOS. Then in Chapter 6 we describe the PAXOS algorithm, prove its correctness and analyze its performance. In Chapter 7 we describe the MULTIPAXOS algorithm. Finally in Chapter 8 we discuss how to use MULTIPAXOS to implement a data replication algorithm. Chapter 9 contains the conclusions.

Chapter 2

Models

In this chapter we describe the I/O automata models we use in this thesis. Section 2.1 presents an overview of the automata models. Then, Section 2.2 describes the basic I/O automaton model, which is used in Section 2.3 to describe the MMT automaton model. Section 2.4 describes the general timed automaton model. In Section 2.5 the Clock GT automaton is introduced; Section 2.5 provides also a technique to transform an MMTA into a Clock GTA. Section 2.6 describes how automata are composed.

2.1 Overview

The I/O automata models are formal models suitable for describing asynchronous and partially synchronous distributed systems. Various I/O automata models have been developed so far (see, for example, [35]). The simplest I/O automata model does not consider time and thus it is suitable for describing asynchronous systems. We remark that in the literature this simple I/O automata model is referred to as the "I/O automaton model". However we prefer to use the general expression "I/O automata models" to indicate all the I/O automata models, henceforth we refer to the simplest one as the "basic I/O automaton model" (BIOA for short). The BIOA model was developed by Lynch and Tuttle [37]. Two extensions of the BIOA model that provide formal mechanisms to handle the passage of time and thus are suitable for describing partially synchronous distributed systems, are the MMT automaton

(MMTA for short), developed by Merritt, Modugno and Tuttle [42] and the general timed automaton (GT automaton or GTA for short) developed by Lynch and Vaandrager [38, 39, 40]. The MMTA is a special case of GTA, and thus it can be regarded as a notation for describing some GT automata.

In this thesis we introduce a particular type of GTA that we call "Clock GTA". The Clock GTA is suitable for describing partially synchronous distributed systems with processors having local clocks; thus it is suitable for describing timing assumptions. In this thesis we use the GTA model and in particular the Clock GT automaton model. However, we use the MMT automaton model to describe some of the Clock GTAs[1]; this is possible because an MMTA is a particular type of GTA; there is a standard technique that transforms an MMTA into a GTA and we specialize this technique in order to transform an MMT automaton into a Clock GTA.

An I/O automaton is a simple type of state machine in which transitions are associated with named *actions*. These actions are classified into categories, namely *input, output, internal* and, for the timed models, *time-passage*. Input and output actions are used for communication with the external environment, while internal actions are local to the automaton. The time-passage actions are intended to model the passage of time. The input actions are assumed not to be under the control of the automaton, that is, they are controlled by the external environment which can force the automaton to execute the input actions. Internal and output actions are controlled by the automaton. The time-passage actions are also controlled by the automaton.

As an example, we can consider an I/O automaton that models the behavior of a process involved in a consensus problem. Figure 2-1 shows the typical interface (that is, input and output actions) of such an automaton. The automaton is drawn as a circle, input actions are depicted as incoming arrows and output actions as outcoming arrows (internal actions are hidden since they are local to the automaton).

[1]The reason why we use MMT automata to describe some of our Clock GT automata is that MMT automata code is simpler. We use MMTA to describe the parts of the algorithm that can run asynchronously and we use the time bounds only for the analysis.

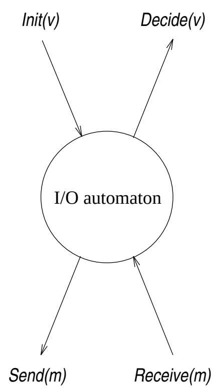

Figure 2-1: An I/O automaton.

The automaton receives inputs from the external world by means of action Init(v), which represents the receipt of an input value v and conveys outputs by means of action Decide(v) which represents a decision of v. Actions Send(m) and Receive(m) are supposed to model the communication with other automata.

2.2 The basic I/O automata model

A *signature* S is a triple consisting of three disjoint sets of actions: the input actions, $in(S)$, the output actions, $out(S)$, and the internal actions, $int(S)$. The external actions, $ext(S)$, are $in(S) \cup out(S)$; the locally controlled actions, $local(S)$, are $out(S) \cup int(S)$; and $acts(S)$ consists of all the actions of S. The external signature, $extsig(S)$, is defined to be the signature $(in(S), out(S), \emptyset)$. The external signature is also referred to as the external interface.

A *basic I/O automaton* (BIOA for short) A, consists of five components:

- $sig(A)$, a signature
- $states(A)$, a (not necessarily finite) set of *states*
- $start(A)$, a nonempty subset of $states(A)$ known as the *start states* or *initial states*
- $trans(A)$, a *state-transition relation*, where $trans(A) \subseteq states(A) \times$

$acts(sig(A)) \times states(A)$; this must have the property that for every state s and every input action π, there is a transition $(s, \pi, s') \in trans(A)$

- $tasks(A)$, a *task partition*, which is an equivalence relation on $local(sig(A))$ having at most countably many equivalence classes

Often $acts(A)$ is used as shorthand for $acts(sig(A))$, and similarly $in(A)$, and so on.

An element (s, π, s') of $trans(A)$ is called a *transition*, or *step*, of A. If for a particular state s and action π, A has some transition of the form (s, π, s'), then we say that π is *enabled* in s. Input actions are enabled in every state.

The fifth component of the I/O automaton definition, the task partition $tasks(A)$, should be thought of as an abstract description of "tasks," or "threads of control," within the automaton. This partition is used to define fairness conditions on an execution of the automaton; roughly speaking, the fairness conditions say that the automaton must continue, during its execution, to give fair turns to each of its tasks.

An *execution fragment* of A is either a finite sequence, $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$, or an infinite sequence, $s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r, \ldots$, of alternating states and actions of A such that $(s_k, \pi_{k+1}, s_{k+1})$ is a transition of A for every $k \geq 0$. Note that if the sequence is finite, it must end with a state. An execution fragment beginning with a start state is called an *execution*. The *length* of a finite execution fragment $\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$ is r. The set of executions of A is denoted by $execs(A)$. A state is said to be *reachable* in A if it is the final state of a finite execution of A.

The *trace* of an execution α of A, denoted by $trace(\alpha)$, is the subsequence of α consisting of all the external actions. A *trace* β of A is a trace β of an execution of A. The set of traces of A is denoted by $traces(A)$.

2.3 The MMT automaton model.

An MMT timed automaton model is obtained simply by adding to the BIOA model lower and upper bounds on the time that can elapse before an enabled action is executed. Formally an MMT automaton consists of a BIOA and a *boundmap* b. A boundmap b is a pair of mappings, *lower* and *upper* which give lower and upper bounds for all the tasks. For each task C, it is required that $0 \leq lower(C) \leq upper(C) \leq \infty$ and that $lower(C) < \infty$. The bounds $lower(C)$ and $upper(C)$ are respectively, a lower bound and an upper bound on the time that can elapse before an enabled action belonging to C is executed.

A *timed execution* of an MMT automaton $B = (A, b)$ is defined to be a finite sequence $\alpha = s_0$, (π_1, t_1), s_1, $(\pi_2, t_2), \ldots, (\pi_r, t_r)$, s_r or an infinite sequence $\alpha = s_0$, (π_1, t_1), s_1, $(\pi_2, t_2), \ldots, (\pi_r, t_r)$, $s_r, \ldots$, where the s's are states of the I/O automaton A, the π's are actions of A, and the t's are times in $\mathbb{R}^{\geq 0}$. It is required that the sequence $s_0, \pi_1, s_1, \ldots$—that is, the sequence α with the times ignored—be an ordinary execution of I/O automaton A. It is also required that the successive times t_r in α be nondecreasing and that they satisfy the lower and upper bound requirements expressed by the boundmap b.

Define r to be an *initial index* for a task C provided that C is enabled in s_r and one of the following is true: (i) $r = 0$; (ii) C is not enabled in s_{r-1}; (iii) $\pi_r \in C$. The initial indices represent the points at which we begin to measure the time bounds of the boundmap. For every initial index r for a task C, it is required that the following conditions hold. (Let $t_0 = 0$.)

Upper bound condition: If there exists $k > r$ with $t_k > t_r + upper(C)$, then there exists $k' > r$ with $t_{k'} \leq t_r + upper(C)$ such that either $\pi_{k'} \in C$ or C is not enabled in $s_{k'}$.

Lower bound condition: There does not exist $k > r$ with $t_k < t_r + lower(C)$ and $\pi_k \in C$.

The upper bound condition says that, from any initial index for a task C, if time ever passes beyond the specified upper bound for C, then in the interim, either an action in C must occur, or else C must become disabled. The lower bound condition says that, from any initial index for C, no action in C can occur before the specified lower bound.

The set of timed executions of B is denoted by $texecs(B)$. A state is said to be *reachable* in B if it is the final state of some finite timed execution of B.

A timed execution is *admissible* provided that the following condition is satisfied:

Admissibility condition: If timed execution α is an infinite sequence, then the times of the actions approach ∞. If α is a finite sequence, then in the final state of α, if task C is enabled, then $upper(C) = \infty$.

The admissibility condition says that time advances normally and that processing does not stop if the automaton is scheduled to perform some more work. The set of admissible timed executions of B is denoted by $atexecs(B)$.

Notice that time bounds of the MMT substitute for the fairness conditions of a BIOA.

The *timed trace* of a timed execution α of B, denoted by $ttrace(\alpha)$, is the subsequence of α consisting of all the external actions, each paired with its associated time. The *admissible timed traces* of B, which are denoted by $attraces(B)$, are the timed traces of admissible timed executions of B.

2.4 The GT automaton model

The GTA model uses *time-passage* actions called $\nu(t)$, $t \in \mathbb{R}^+$ to model the passage of time. The time-passage action $\nu(t)$ represents the passage of time by the amount t.

A *timed signature* S is a quadruple consisting of four disjoint sets of actions: the input actions $in(S)$, the output actions $out(S)$, the internal actions $int(S)$, and the time-passage actions. For a GTA

- the *visible actions*, $vis(S)$, are the input and output actions, $in(S) \cup out(S)$
- the *external actions*, $ext(S)$, are the visible and time-passage actions, $vis(S) \cup \{\nu(t) : t \in \mathbb{R}^+\}$
- the *discrete actions*, $disc(S)$, are the visible and internal actions, $vis(S) \cup int(S)$
- the *locally controlled actions*, $local(S)$, are the output and internal actions, $out(S) \cup int(S)$

- $acts(S)$ are all the actions of S

A GTA A consists of the following four components:

- $sig(A)$, a timed signature
- $states(A)$, a set of *states*
- $start(A)$, a nonempty subset of $states(A)$ known as the *start states* or *initial states*
- $trans(A)$, a *state transition relation*, where $trans(A) \subseteq states(A) \times acts(sig(A)) \times states(A)$; this must have the property that for every state s and every input action π, there is a transition $(s, \pi, s') \in trans(A)$

Often $acts(A)$ is used as shorthand for $acts(sig(A))$, and similarly $in(A)$, and so on.

An element (s, π, s') of $trans(A)$ is called a *transition*, or *step*, of A. If for a particular state s and action π, A has some transition of the form (s, π, s'), then we say that π is *enabled* in s. Since every input action is required to be enabled in every state, automata are said to be *input-enabled.* The input-enabling assumption means that the automaton is not able to somehow "block" input actions from occurring.

There are two simple axioms that A is required to satisfy:

A1: If $(s, \nu(t), s')$ and $(s', \nu(t'), s'')$ are in $trans(A)$, then $(s, \nu(t+t'), s'')$ is in $trans(A)$.

A2: If $(s, \nu(t), s') \in trans(A)$ and $0 < t' < t$, then there is a state s'' such that $(s, \nu(t'), s'')$ and $(s'', \nu(t - t'), s')$ are in $trans(A)$.

Axiom A1 allows repeated time-passage steps to be combined into one step, while Axiom A2 is a kind of converse to A1 that allows a time-passage step to be split in two.

A *timed execution fragment* of a GTA, A, is defined to be either a finite sequence $\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$ or an infinite sequence $\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r, \ldots$, where the s's are states of A, the π's are actions (either input, output, internal, or time-passage) of A, and $(s_k, \pi_{k+1}, s_{k+1})$ is a step (or transition) of A for every k. Note that if the sequence is finite, it must end with a state. The length of a finite execution fragment $\alpha = s_0, \pi_1, s_1, \pi_2, \ldots, \pi_r, s_r$ is r. A timed execution fragment beginning with a start state is called a *timed execution*.

Axioms A1 and A2, say that there is not much difference between timed execution fragments that differ only by splitting and combining time-passage steps. Two timed execution fragments α and α' are *time-passage equivalent* if α can be transformed into α' by splitting and combining time-passage actions according to Axioms A1 and A2.

If α is any timed execution fragment and π_r is any action in α, then we say that the *time of occurrence* of π_r is the sum of all the reals in the time-passage actions preceding π_r in α. A timed execution fragment α is said to be *admissible* if the sum of all the reals in the time-passage actions in α is ∞. The set of admissible timed executions of A is denoted by $atexecs(A)$. A state is said to be *reachable* in A if it is the final state of a finite timed execution of A.

The *timed trace* of a timed execution fragment α, denoted by $ttrace(\alpha)$, is the sequence of visible events in α, each paired with its time of occurrence. The *admissible timed traces* of A, denoted by $attraces(A)$, are the timed traces of admissible timed executions of A.

We may refer to a timed execution simply as an execution. Similarly a timed trace can be referred to as a trace.

2.5 The Clock GT automaton model

A Clock GTA is a GTA with a special component included in the state; this special variable is called *Clock* and it can assume values in $\mathbb{R}$. The purpose of *Clock* is to model the local clock of the process. The only actions that are allowed to modify *Clock* are the time-passage actions $\nu(t)$. When a time-passage action $\nu(t)$ is executed by an automaton, the *Clock* is incremented by an amount of time $t' \geq 0$ independent of the amount t of time specified by the time-passage action.

Formally, beside Axioms A1 and A2, of a regular GTA, for a Clock GTA we require that

A3: If $(s, \nu(t), s')$ is in $trans(\mathrm{A})$, then $(s, \nu(\tilde{t}), s')$, for any $\tilde{t} > 0$, is in $trans(\mathrm{A})$.

Hence a Clock GTA cannot keep track of the real time. Since the occurrence of the time-passage action $\nu(t)$ represents the passage of (real) time by the amount t, by incrementing the variable *Clock* by an amount t' different from t we are able to model the passage of clock time by the amount t'. As a special case, we have some time-passage actions in which $t' = t$; in these cases the local clock of the process is running at the speed of real time.
In the following and in the rest of the thesis, we use the notation $s.x$ to denote the value of state component x in state s.

Definition 2.5.1 *A step $(s_{k-1}, \nu(t), s_k)$ of a Clock GTA is called regular if $s_k.Clock - s_{k-1}.Clock = t$; it is called irregular if it is not regular.*

That is, a time-passage step executing action $\nu(t)$ is regular if it increases *Clock* by $t' = t$. In a regular time-passage step, the local clock is increased by the same amount as the real time, whereas in an irregular time-passage step $\nu(t)$ that represents the passage of real time by the amount t, the local clock is increased either by $t' < t$ (the local clock is slower than the real time) or by $t' > t$ (the local clock is faster than the real time).

Definition 2.5.2 *A timed execution fragment α of a Clock GTA is called regular if all the time-passage steps of α are regular. It is called irregular if it is not regular, i.e., if at least one of its time-passage steps is irregular.*

In a partially synchronous distributed system processes are expected to respond and messages are expected to be delivered within given time bounds. A timing failure is a violation of these time bounds. An irregular time-passage step can model the occurrence of a timing failure in the following way. Assume that a particular process is supposed to execute some action within a given time bound and this time bound is modeled by restricting the time that can elapse with time-passage actions (this is how GT automata are used to model partial synchrony). We can impose the same time bound on the clock time so that when time-passage action $\nu(t)$ is executed the local clock of the particular process we are considered is actually incremented by an amount t' so that the time bound on the local clock time is not violated. Clearly if $t' < t$ then it is possible that a violation of the time bound (with respect to real time) actually occurs.

We remark that a time bound violation can actually be either an upper bound violation (a process or a channel is slower than expected) or a lower bound violation (a process or a channel is faster than expected). The latter can be modeled with a time-passage action $\nu(t)$ that increments the clock time by an amount t', $t' > t$.

Though we have defined a regular execution fragment so that it does not contain any of the timing failures, we remark that for the the scope of this thesis we actually need only that the former type of timing failures (upper bound) does not happen. That is, for the scope of this thesis, we could have defined a regular step $\nu(t)$ as one that increases the clock time by an amount t', $t' \geq t$. However, to not lose in generality, a regular step is defined to rule out both kinds timing failures. Obviously, in a regular execution fragment there are no timing failures.

Transforming MMTA into Clock GTA. MMT automata are a special case of GT automata. There is a standard transformation technique that given an MMTA produces an equivalent GTA, i.e., one that has the same external behavior (see Section 23.2.2 of [35]).

Next, we show how to transform any MMT automaton (A, b) into an equivalent clock general timed automaton $A' = clockgen(A, b)$. Automaton A' acts like automaton A, but the time bounds of the boundmap b are expressed as restrictions on the value that the local time can assume. The technique used is essentially the same as the one that transforms an MMTA into an equivalent GTA with some modifications to handle the *Clock* variable.

The transformation involves building clock time deadlines into the state and not allowing the clock time to pass beyond those deadlines while they are still in force.

The deadlines are set according to the boundmap b. New constraints on non-time-passage actions are added to express the lower bound conditions. Notice however, that all these constraints are on the clock time, while in the transformation of an MMTA into a GTA they are on the real time.

More specifically, the state of the underlying BIOA A is augmented with a *Clock* component, plus *First*(C) and *Last*(C) components for each task C. The *First*(C) and *Last*(C) components represent, respectively, the earliest and latest clock times at which the next action in task C is allowed to occur. The time-passage actions $\nu(t)$ are also added.

The *First* and *Last* components get updated by the various steps, according to the *lower* and *upper* bounds specified by the boundmap b. The time-passage actions $\nu(t)$ have an explicit precondition saying that the clock time cannot pass beyond any of the *Last*(C) values; this is because these represent deadlines for the various tasks. Restrictions are also added on actions in any task C, saying that the current clock time *Clock* must be at least as great as the lower bound *First*(C).

In more detail, the timed signature of $A' = clockgen(A, b)$ is the same as the signature of A, with the addition of the time-passage actions $\nu(t)$, $t \in \mathbb{R}^+$. Each state of A' consists of the following components:

$basic \in states(A)$, initially a start state of A

$Clock \in \mathbb{R}$, initially arbitrary

For each task C of A:

$First(C) \in \mathbb{R}$, initially $Clock + lower(C)$ if C is enabled in state $basic$, otherwise 0

$Last(C) \in \mathbb{R} \cup \{\infty\}$, initially $Clock + upper(C)$ if C is enabled in $basic$, otherwise ∞

The transitions are defined as follows. If $\pi \in acts(A)$, then $(s, \pi, s') \in trans(A')$ exactly if all the following conditions hold:

1. $(s.basic, \pi, s'.basic) \in trans(A)$.

2. $s'.\mathit{Clock} = s.\mathit{Clock}$.

3. For each $C \in \mathit{tasks}(A)$,

   (a) If $\pi \in C$, then $s.\mathit{First}(C) \leq s.\mathit{Clock}$.

   (b) If C is enabled in both $s.\mathit{basic}$ and $s'.\mathit{basic}$ and $\pi \notin C$, then $s.\mathit{First}(C) = s'.\mathit{First}(C)$ and $s.\mathit{Last}(C) = s'.\mathit{Last}(C)$.

   (c) If C is enabled in $s'.\mathit{basic}$ and either C is not enabled in $s.\mathit{basic}$ or $\pi \in C$, then $s'.\mathit{First}(C) = s.\mathit{Clock} + \mathit{lower}(C)$ and $s'.\mathit{Last}(C) = s.\mathit{Clock} + \mathit{upper}(C)$.

   (d) If C is not enabled in $s'.\mathit{basic}$, then $s'.\mathit{First}(C) = 0$ and $s'.\mathit{Last}(C) = \infty$.

If $\pi = \nu(t)$, then $(s, \pi, s') \in \mathit{trans}(A')$ exactly if all the following conditions hold:

1. $s'.\mathit{basic} = s.\mathit{basic}$.

2. $s'.\mathit{Clock} \geq s.\mathit{Clock}$.

3. For each $C \in \mathit{tasks}(A)$,

   (a) $s'.\mathit{Clock} \leq s.\mathit{Last}(C)$.

   (b) $s'.\mathit{First}(C) = s.\mathit{First}(C)$ and $s'.\mathit{Last}(C) = s.\mathit{Last}(C)$.

It should be clear that the above transformation yields a Clock GTA and that for each task C, the lower and upper time bounds specified by the boundmap's components $\mathit{First}(C)$ and $\mathit{Last}(C)$ are transformed into lower and upper bounds on the clock time. More formally we have the following lemma.

Lemma 2.5.3 *In any reachable state of $\mathit{clockgen}(A, b)$ and for any task C of A, we have that.*

1. *$\mathit{Clock} \leq \mathit{Last}(C)$.*

2. *If C is enabled, then $\mathit{Last}(C) \leq \mathit{Clock} + \mathit{upper}(C)$.*

3. *$\mathit{First}(C) \leq \mathit{Clock} + \mathit{lower}(C)$.*

4. *First*$(C) \leq$ *Last*(C).

If some of the timing requirements specified by b are trivial—that is, if some lower bounds are 0 or some upper bounds are ∞—then it is possible to simplify the automaton $clockgen(A, b)$ just by omitting mention of these components. In this thesis all the MMT automata have boundmaps that specify a lower bound of 0 and an upper bound of a fixed constant ℓ; thus the above general transformation could be simplified (by omitting mention of $First(C)$ and using ℓ instead of $upper(C)$, for any C) for our purposes. In the following lemma we assume $lower(C) = 0$ and $upper(C) = \ell$.

Informally the lemma states that if $B = clockgen(A, b)$ is a clock GTA obtained by transforming an MMTA A with boundmap b, then the upper bounds on tasks of A specified by the boundmap b are not violated when B executes a regular execution fragment.

Lemma 2.5.4 *Consider a regular execution fragment α of $clockgen(A, b)$, starting from a reachable state s_0 and lasting for more than ℓ time. Assume that $lower(C) = 0$ and $upper(C) = \ell$ for each task C of automaton A. Then (i) any task C enabled in s_0 either has a step or is disabled within ℓ time, and (ii) any new enabling of C has a subsequent step or disabling within ℓ time, provided that α lasts for more than ℓ time from the enabling of C.*

Proof: Let us first prove (i). Let C be a task enabled in state s_0. By Lemma 2.5.3 we have that $s_0.First(C) \leq s_0.Clock \leq s_0.Last(C)$ and that $s_0.Last(C) \leq s_0.Clock + \ell$. Since the execution is regular and lasts for more then ℓ time, within time ℓ, *Clock* passes the value $s_0.Clock{+}\ell$. But this cannot happen (since $s_0.Last(C) \leq s_0.Clock{+}\ell$) unless $Last(C)$ is increased, which means either C has a step or it is disabled within ℓ time. The proof of (ii) is similar. Let s be the state in which C becomes enabled. Then the proof is as before replacing s_0 with s. ■

2.6 Composition of automata

The composition operation allows an automaton representing a complex system to be constructed by composing automata representing simpler system components. The most important characteristic of the composition of automata is that properties of isolated system components still hold when those isolated components are composed with other components. The composition identifies actions with the same name in different component automata. When any component automaton performs a step involving π, so do all component automata that have π in their signatures.

Since internal actions of an automaton A are intended to be unobservable by any other automaton B, automaton A cannot be composed with automaton B unless the internal actions of A are disjoint from the actions of B. Moreover, A and B cannot be composed unless the sets of output actions of A and B are disjoint.

2.6.1 Composition of BIOA.

Let I be an arbitrary finite index set[2]. A finite countable collection $\{S_i\}_{i \in I}$ of signatures is said to be *compatible* if for all $i, j \in I$, $i \neq j$, the following hold[3]:

1. $int(S_i) \cap acts(S_j) = \emptyset$
2. $out(S_i) \cap out(S_j) = \emptyset$

A finite collection of automata is said to be *compatible* if their signatures are compatible.

When we compose a collection of automata, output actions of the components become output actions of the composition, internal actions of the components become internal actions of the composition, and actions that are inputs to some components

[2]The composition operation for BIOA is defined also for an infinite but countable collection of automata [35], but we only consider the composition of a finite number of automata.

[3]We remark that for the composition of an infinite countable collection of automata, there is a third condition on the definition of compatible signature [35]. However this third condition is automatically satisfied when considering only finite sets of automata.

but outputs of none become input actions of the composition. Formally, the *composition* $S = \prod_{i \in I} S_i$ of a finite compatible collection of signatures $\{S_i\}_{i \in I}$ is defined to be the signature with

- $out(S) = \cup_{i \in I} out(S_i)$
- $int(S) = \cup_{i \in I} int(S_i)$
- $in(S) = \cup_{i \in I} in(S_i) - \cup_{i \in I} out(S_i)$

The *composition* $A = \prod_{i \in I} A_i$ of a finite collection of automata, is defined as follows:[4]

- $sig(A) = \prod_{i \in I} sig(A_i)$
- $states(A) = \prod_{i \in I} states(A_i)$
- $start(A) = \prod_{i \in I} start(A_i)$
- $trans(A)$ is the set of triples (s, π, s') such that, for all $i \in I$, if $\pi \in acts(A_i)$, then $(s_i, \pi, s'_i) \in trans(A_i)$; otherwise $s_i = s'_i$
- $tasks(A) = \cup_{i \in I} tasks(A_i)$

Thus, the states and start states of the composition automaton are vectors of states and start states, respectively, of the component automata. The transitions of the composition are obtained by allowing all the component automata that have a particular action π in their signature to participate simultaneously in steps involving π, while all the other component automata do nothing. The task partition of the composition's locally controlled actions is formed by taking the union of the components' task partitions; that is, each equivalence class of each component automaton becomes an equivalence class of the composition. This means that the task structure of individual components is preserved when the components are composed. Notice

[4]The Π notation in the definition of $start(A)$ and $states(A)$ refers to the ordinary Cartesian product, while the Π notation in the definition of $sig(A)$ refers to the composition operation just defined, for signatures. Also, the notation s_i denotes the ith component of the state vector s.

that since the automata A_i are input-enabled, so is their composition. The following theorem follows from the definition of composition.

Theorem 2.6.1 *The composition of a compatible collection of BIO automata is a BIO automaton.*

The following theorems relate the executions and traces of a composition to those of the component automata. The first says that an execution or trace of a composition "projects" to yield executions or traces of the component automata. Given an execution, $\alpha = s_0, \pi_1, s_1, \dots$, of A, let $\alpha|A_i$ be the sequence obtained by deleting each pair π_r, s_r for which π_r is not an action of A_i and replacing each remaining s_r by $(s_r)_i$, that is, automaton A_i's piece of the state s_r. Also, given a trace β of A (or, more generally, any sequence of actions), let $\beta|A_i$ be the subsequence of β consisting of all the actions of A_i in β. Also, $|$ represents the subsequence of a sequence β of actions consisting of all the actions in a given set in β.

Theorem 2.6.2 *Let $\{A_i\}_{i\in I}$ be a compatible collection of automata and let $A = \prod_{i\in I} A_i$.*

1. *If $\alpha \in execs(A)$, then $\alpha|A_i \in execs(A_i)$ for every $i \in I$.*
2. *If $\beta \in traces(A)$, then $\beta|A_i \in traces(A_i)$ for every $i \in I$.*

The other two are converses of Theorem 2.6.2. The next theorem says that, under certain conditions, executions of component automata can be "pasted together" to form an execution of the composition.

Theorem 2.6.3 *Let $\{A_i\}_{i\in I}$ be a compatible collection of automata and let $A = \prod_{i\in I} A_i$. Suppose α_i is an execution of A_i for every $i \in I$, and suppose β is a sequence of actions in $ext(A)$ such that $\beta|A_i = trace(\alpha_i)$ for every $i \in I$. Then there is an execution α of A such that $\beta = trace(\alpha)$ and $\alpha_i = \alpha|A_i$ for every $i \in I$.*

The final theorem says that traces of component automata can also be pasted together to form a trace of the composition.

Theorem 2.6.4 *Let $\{A_i\}_{i\in I}$ be a compatible collection of automata and let $A = \prod_{i\in I} A_i$. Suppose β is a sequence of actions in $ext(A)$. If $\beta|A_i \in traces(A_i)$ for every $i \in I$, then $\beta \in traces(A)$.*

Theorem 2.6.4 implies that in order to show that a sequence is a trace of a system, it is enough to show that its projection on each individual system component is a trace of that component.

2.6.2 Composition of MMTA.

MMT automata can be composed in much the same way as BIOA, by identifying actions having the same name in different automata.

Let I be an arbitrary finite index set. A finite collection of MMT automata is said to be be *compatible* if their underlying BIO automata are compatible. Then the *composition* $(A, b) = \prod_{i\in I}(A_i, b_i)$ of a finite compatible collection of MMT automata $\{(A_i, b_i)\}_{i\in I}$ is the MMT automaton defined as follows:

- $A = \prod_{i\in I} A_i$, that is, A is the composition of the underlying BIO automata A_i for all the components.
- For each task C of A, b's *lower* and *upper* bounds for C are the same as those of b_i, where A_i is the unique component I/O automaton having task C.

Clearly we have the following theorem.

Theorem 2.6.5 *The composition of a compatible collection of MMT automata is an MMT automaton.*

The following theorems correspond to Theorems 2.6.2–2.6.4 stated for BIOA.

Theorem 2.6.6 *Let $\{B_i\}_{i\in I}$ be a compatible collection of MMT automata and let $B = \prod_{i\in I} B_i$.*

1. *If $\alpha \in atexecs(B)$, then $\alpha|B_i \in atexecs(B_i)$ for every $i \in I$.*
2. *If $\beta \in attraces(B)$, then $\beta|B_i \in attraces(B_i)$ for every $i \in I$.*

Theorem 2.6.7 *Let $\{B_i\}_{i\in I}$ be a compatible collection of MMT automata and let $B = \prod_{i\in I} B_i$. Suppose α_i is an admissible timed execution of B_i for every $i \in I$ and suppose β is a sequence of (action,time) pairs, where all the actions in β are in $ext(A)$, such that $\beta|B_i = ttrace(\alpha_i)$ for every $i \in I$. Then there is an admissible timed execution α of B such that $\beta = ttrace(\alpha)$ and $\alpha_i = \alpha|B_i$ for every $i \in I$.*

Theorem 2.6.8 *Let $\{B_i\}_{i\in I}$ be a compatible collection of MMT automata and let $B = \prod_{i\in I} B_i$. Suppose β is a sequence of (action,time) pairs, where all the actions in β are in $ext(A)$. If $\beta|B_i \in attraces(B_i)$ for every $i \in I$, then $\beta \in attraces(B)$.*

2.6.3 Composition of GTA.

Let I be an arbitrary finite index set. A finite collection $\{S_i\}_{i\in I}$ of timed signatures is said to be *compatible* if for all $i, j \in I$, $i \neq j$, we have

1. $int(S_i) \cap acts(S_j) = \emptyset$
2. $out(S_i) \cap out(S_j) = \emptyset$

A collection of GTAs is *compatible* if their timed signatures are compatible.

The *composition* $S = \prod_{i\in I} S_i$ of a finite compatible collection of timed signatures $\{S_i\}_{i\in I}$ is defined to be the timed signature with

- $out(S) = \cup_{i\in I} out(S_i)$
- $int(S) = \cup_{i\in I} int(S_i)$
- $in(S) = \cup_{i\in I} in(S_i) - \cup_{i\in I} out(S_i)$

The *composition* $A = \prod_{i\in I} A_i$ of a finite compatible collection of GTAs $\{A_i\}_{i\in I}$ is defined as follows:

- $sig(A) = \prod_{i\in I} sig(A_i)$
- $states(A) = \prod_{i\in I} states(A_i)$
- $start(A) = \prod_{i\in I} start(A_i)$

- $trans(A)$ is the set of triples (s, π, s') such that, for all $i \in I$, if $\pi \in acts(A_i)$, then $(s_i, \pi, s'_i) \in trans(A_i)$; otherwise $s_i = s'_i$

The transitions of the composition are obtained by allowing all the components that have a particular action π in their signature to participate, simultaneously, in steps involving π, while all the other components do nothing. Note that this implies that all the components participate in time-passage steps, with the same amount of time passing for all of them.

Theorem 2.6.9 *The composition of a compatible collection of general timed automata is a general timed automaton.*

The following theorems correspond to Theorems 2.6.2–2.6.4 stated for BIOA and to Theorems 2.6.6–2.6.8 stated for MMTA. Theorem 2.6.11, has a small technicality that is a consequence of the fact that the GTA model allows consecutive time-passage steps to appear in an execution. Namely, the admissible timed execution α that is produced by "pasting together" individual admissible timed executions α_i might not project to give exactly the original α_i's, but rather admissible timed executions that are time-passage equivalent to the original α_i's.

Theorem 2.6.10 *Let $\{B_i\}_{i \in I}$ be a compatible collection of general timed automata and let $B = \prod_{i \in I} B_i$.*

1. *If $\alpha \in atexecs(B)$, then $\alpha | B_i \in atexecs(B_i)$ for every $i \in I$.*

2. *If $\beta \in attraces(B)$, then $\beta | B_i \in attraces(B_i)$ for every $i \in I$.*

Theorem 2.6.11 *Let $\{B_i\}_{i \in I}$ be a compatible collection of general timed automata and let $B = \prod_{i \in I} B_i$. Suppose α_i is an admissible timed execution of B_i for every $i \in I$, and suppose β is a sequence of (action,time) pairs, with all the actions in $vis(B)$, such that $\beta | B_i = ttrace(\alpha_i)$ for every $i \in I$. Then there is an admissible timed execution α of B such that $\beta = ttrace(\alpha)$ and α_i is time-passage equivalent to $\alpha | B_i$ for every $i \in I$.*

Theorem 2.6.12 *Let $\{B_i\}_{i\in I}$ be a compatible collection of general timed automata and let $B = \prod_{i\in I} B_i$. Suppose β is a sequence of (action,time) pairs, where all the actions in β are in $vis(A)$. If $\beta|B_i \in attraces(B_i)$ for every $i \in I$, then $\beta \in attraces(B)$.*

2.6.4 Composition of Clock GTA.

Clock GT automata are GT automata; thus, they can be composed as GT automata are composed. However we point out that the composition of Clock GT automata does not yield a Clock GTA but a GTA. This follows from the fact that in a composition of Clock GT automata there is more than one special state component *Clock*. It is possible to generalize the definition of Clock GTA by letting a Clock GTA have several special state components $Clock_1, Clock_2, \ldots$ so that the composition of Clock GT automata is still a Clock GTA. However we do not make this extension in this thesis, since for our purposes we do not need the composition of Clock GT automata to be a Clock GTA.

2.7 Bibliographic notes

The basic I/O automata was introduced by Lynch and Tuttle in [37]. The MMT automaton model was designed by Merritt, Modugno, and Tuttle [42]. Work on transformation of MMT automata to GT automata has been done by Lynch and Attiya [36]. The definition used in this chapter is the one provided in the book by Lynch [35]. The GT automaton model was introduced by Lynch and Vaandrager [38, 39, 40], though they did not use the name General Timed Automaton. The book by Lynch [35] contains a broad coverage of these models and more pointers to the relevant literature.

Chapter 3

The distributed setting

In this chapter we discuss the distributed setting. We consider a complete network of n processes communicating by exchange of messages in a partially synchronous setting. Each process of the system is uniquely identified by its identifier $i \in \mathcal{I}$, where $\mathcal{I}$ is a totally ordered finite set of n identifiers. The set $\mathcal{I}$ is known by all the processes. Moreover each process of the system has a local clock. Local clocks can run at different speeds, though in general we expect them to run at the same speed as real time. We assume that a local clock is available also for channels; though this may seem somewhat strange, it is just a formal way to express the fact that a channel is able to deliver a given message within a fixed amount of time, by relying on some timing mechanism (which we model with the local clock). We use Clock GT automata to model both processes and channels.

Throughout the thesis we use two constants, ℓ and d, to represent upper bounds on the time needed to execute an enabled process action and to deliver a message, respectively. These bounds do not necessarily hold for every action and message in every execution; a violation of these bounds is a timing failure. A Clock GTA models timing failures with irregular time-passage actions.

3.1 Processes

A process is modeled by a Clock GT automaton. We allow process stopping failures and recoveries and timing failures. To formally model process stops and recoveries we model process i with a Clock GTA which has a special state component called $Status_i$ and two input actions Stop_i and Recover_i. The state variable $Status_i$ reflects the current condition of process i. The effect of action Stop_i is to set $Status_i$ to `stopped`, while the effect of Recover_i is to set $Status_i$ to `alive`. Moreover when $Status_i = \texttt{stopped}$, all the locally controlled actions are not enabled and the input actions have no effect, except for action Recover_i.

Definition 3.1.1 *A "process automaton" for process i is a Clock GTA having the special $Status_i$ variable and input actions $Stop_i$ and $Recover_i$ and whose behavior satisfies the following. The effect of action $Stop_i$ is to set $Status_i$ to* `stopped`*, while the effect of $Recover_i$ is to set $Status_i$ to* `alive`*. In any reachable state s such that $s.Status = \texttt{stopped}$ the only possible steps are (s, π, s') where π is an input action. Moreover when $s.Status = \texttt{stopped}$ for all $\pi \neq Recover_i$ state s' is equal to state s. A process automaton has an upper bound of ℓ on the clock time that can elapse before an enabled action is executed.*

We remark that the time bound is directly encoded into the steps of process automata. When the execution is regular, the local clock runs at the speed of real time and thus the time bound holds with respect to the real time, too.

For simplicity we refer to a "process automaton" simple as a "process". A process can be alive or dead in a state or an execution fragment of an execution. More formally:

Definition 3.1.2 *We say that a process i is alive (resp. stopped) in a given state of an execution of a system which includes process i, if in that state we have $Status_i =$* `alive` *(resp. $Status_i = \texttt{stopped}$).*

Definition 3.1.3 *We say that a process i is alive (resp. stopped) in a given execution fragment of an execution of a system which includes process i, if it is alive (resp.*

stopped) in all the states of the execution fragment.

Between a failure and a recovery a process does not lose its state. We remark that PAXOS needs only a small amount of stable storage (see Section 6.6); however, for simplicity, we assume that the entire state of a process is stable.

Finally, we provide the following definition of "stable" execution fragment of a given process. This definition will be used later to define a stable execution of a distributed system.

Definition 3.1.4 *Given a process automaton* PROCESS$_i$*, we say that an execution fragment* α *of* PROCESS$_i$ *is "stable" if process* i *is either stopped or alive in* α *and* α *is regular.*

3.2 Channels

We consider unreliable channels that can lose and duplicate messages. Reordering of messages is not considered a failure. Timing failures are also possible. Figure 3-1 shows the code of a Clock GT automaton CHANNEL$_{i,j}$, which models the communication channel from process i to process j; there is one automaton for each possible choice of i and j. Notice that we allow the possibility that $i = j$, that is, the sender and the receiver are the same process. We denote by $\mathcal{M}$ the set of messages that can be sent over the channels. The interface of CHANNEL$_{i,j}$, consists of input actions modelling failures, namely Lose$_{i,j}$ and Duplicate$_{i,j}$, and input actions Send$(m)_{i,j}$, $m \in \mathcal{M}$, which are used by process i to send messages to process j, and output actions Receive$(m)_{i,j}$, $m \in \mathcal{M}$, which are used by the channel automaton to deliver messages sent by process i to process j.

Channel failures are formally modeled as input actions Lose$_{i,j}$, and Duplicate$_{i,j}$. The effect of these two actions is to manipulate *Msgs*. In particular Lose$_{i,j}$ deletes one message from *Msgs*; Duplicate$_{i,j}$ duplicates one of the messages in *Msgs*. When the execution is regular, automaton CHANNEL$_{i,j}$ guarantees that messages are delivered within time d of the sending. When the execution is irregular, messages can take arbitrarily long time to be delivered.

$\text{CHANNEL}_{i,j}$

Signature:

Input:	$\text{Send}(m)_{i,j}$, $\text{Lose}_{i,j}$, $\text{Duplicate}_{i,j}$
Output:	$\text{Receive}(m)_{i,j}$
Time-passage:	$\nu(t)$

State:

Clock $\in \mathbb{R}$, initially arbitrary
Msgs, a set of elements of $\mathcal{M} \times \mathbb{R}$, initially empty

Actions:

input $\text{Send}(m)_{i,j}$
Eff: add (m, Clock) to *Msgs*

output $\text{Receive}(m)_{i,j}$
Pre: (m, t) is in *Msgs*, for some t
Eff: remove (m, t) from *Msgs*

time-passage $\nu(t)$
Pre: Let $t' \geq 0$ be such that for all $(m, t'') \in \mathit{Msgs}$, $\mathit{Clock} + t' \leq t'' + d$
Eff: $\mathit{Clock} := \mathit{Clock} + t'$

input $\text{Lose}_{i,j}$
Eff: remove one element of *Msgs*

input $\text{Duplicate}_{i,j}$
Eff: let (m, t) be an element of *Msgs*
let t' such that $t \leq t' \leq \mathit{Clock}$
place (m, t') into *Msgs*

Figure 3-1: Automaton $\text{CHANNEL}_{i,j}$

The next lemma provides a basic property of $\textsc{channel}_{i,j}$.

Lemma 3.2.1 *In a reachable state s of $\textsc{channel}_{i,j}$, if a message $(m,t) \in s.Msgs_{i,j}$ then $t \leq s.Clock_{i,j} \leq t+d$.*

Proof: We prove the lemma by induction on the length k of an execution $\alpha = s_0\pi_1 s_1 \ldots\ s_{k-1}\pi_k s_k$. The base $k=0$ is trivial since $s_0.Msgs$ is empty. For the inductive step assume that the assertion is true in state s_k and consider the execution $\alpha\pi s$. We need to prove that the assertion is still true in s. Actions $\text{Lose}_{i,j}$, $\text{Duplicate}_{i,j}$, and $\text{Receive}(m)_{i,j}$, do not add any new element to *Msgs* and do not modify *Clock*; hence they cannot make the assertion false. Thus we only need to consider the cases $\pi = \text{Send}(m)_{i,j}$ and $\pi = \nu(t)$. If $\pi = \text{Send}(m)_{i,j}$ a new element (m,t), with $t = s_k.Clock$ is added to *Msgs*; however since $s_k.Clock = s.Clock$ the assertion is still true in state s. If $\pi = \nu(t)$, by the precondition of $\nu(t)$, we have that $s.Clock \leq t+d$ for all (m,t) in *Msgs*. Thus the assertion is true also in state s. ■

We remark that if $\textsc{channel}_{i,j}$ is not in a reachable state then it may be unable to take time-passage steps, because $Msgs_{i,j}$ may contain messages (m,t) for which $Clock_{i,j} > t+d$ and thus the time-passage actions are no longer enabled, that is, time cannot pass.

The following definition of "stable" execution fragment for a channel captures the condition under which messages are delivered on time.

Definition 3.2.2 *Given a channel $\textsc{channel}_{i,j}$, we say that an execution fragment α of $\textsc{channel}_{i,j}$ is "stable" if no $Lose_{i,j}$ and $Duplicate_{i,j}$ actions occur in α and α is regular.*

Next lemma proves that in a stable execution fragment messages are delivered within time d of the sending.

Lemma 3.2.3 *In a stable execution fragment α of $\textsc{channel}_{i,j}$ beginning in a reachable state s and lasting for more than d time, we have that (i) all messages (m,t) that in state s are in $Msgs_{i,j}$ are delivered by time d, and (ii) any message sent in α*

is delivered within time d of the sending, provided that α lasts for more than d time from the sending of the message.

Proof: Let us first prove assertion (i). Let (m, t) be a message belonging to $s.Msgs_{i,j}$. By Lemma 3.2.1 we have that $t \leq s.Clock_{i,j} \leq t + d$. However since α is stable, the time-passage actions increment $Clock_{i,j}$ at the speed of real time and since α lasts for more than d time, $Clock$ passes the value $t + d$. However this cannot happen if m is not delivered since by the preconditions of $\nu(t)$ of $\text{CHANNEL}_{i,j}$, all the increments t' of $Clock_{i,j}$ are such that $Clock_{i,j} + t' \leq t + d$. Notice that m cannot be lost (by a $\text{Lose}_{i,j}$ action), since α is stable.

Now let us prove assertion (ii). Let $(s', \text{Send}(m)_{i,j}, s'')$ be the step that puts (m, t), with $t = s'.Clock$, in $Msgs$. Since $s'.Clock = s''.Clock$, we have that $s''.Clock_{i,j} = t$. Since α is stable, the time-passage actions increment $Clock_{i,j}$ at the speed of real time and since α lasts for more than d time from the sending of m, $Clock_{i,j}$ passes the value $t + d$. However this cannot happen if m is not delivered since by the preconditions of $\nu(t)$ of $\text{CHANNEL}_{i,j}$, all the increments t' of $Clock_{i,j}$ are such that $Clock_{i,j} + t' \leq t + d$. Again, notice that m cannot be lost (by a $\text{Lose}_{i,j}$ action), since α is stable. ■

3.3 Distributed systems

In this section we give a formal definition of distributed system. A distributed system is the composition of automata modelling channels and processes. We are interested in modelling bad and good behaviors of a distributed system; in order to do so we provide some definitions that characterize the behavior of a distributed system. The definition of "nice" execution fragment given in the following captures the good behavior of a distributed system. Informally, a distributed system behaves nicely if there are no process failures and recoveries, no channel failures and no irregular steps—remember that an irregular step models a timing failure—and a majority of the processes are alive.

Definition 3.3.1 *Given a set $\mathcal{J} \subseteq \mathcal{I}$ of processes, a communication system for $\mathcal{J}$ is* *the composition of channel automata* $\text{CHANNEL}_{i,j}$ *for all possible choices of* $i, j \in \mathcal{J}$.

Definition 3.3.2 *A distributed system is the composition of process automata modeling some set* $\mathcal{J}$ *of processes and a communication system for* $\mathcal{J}$.

In this thesis we will always compose automata that model the set of all processes $\mathcal{I}$. Thus we define the communication system S_{CHA} to be the communication system for the set $\mathcal{I}$ of all processes. Figure 3-2 shows this communication system and its interactions with the external environment.

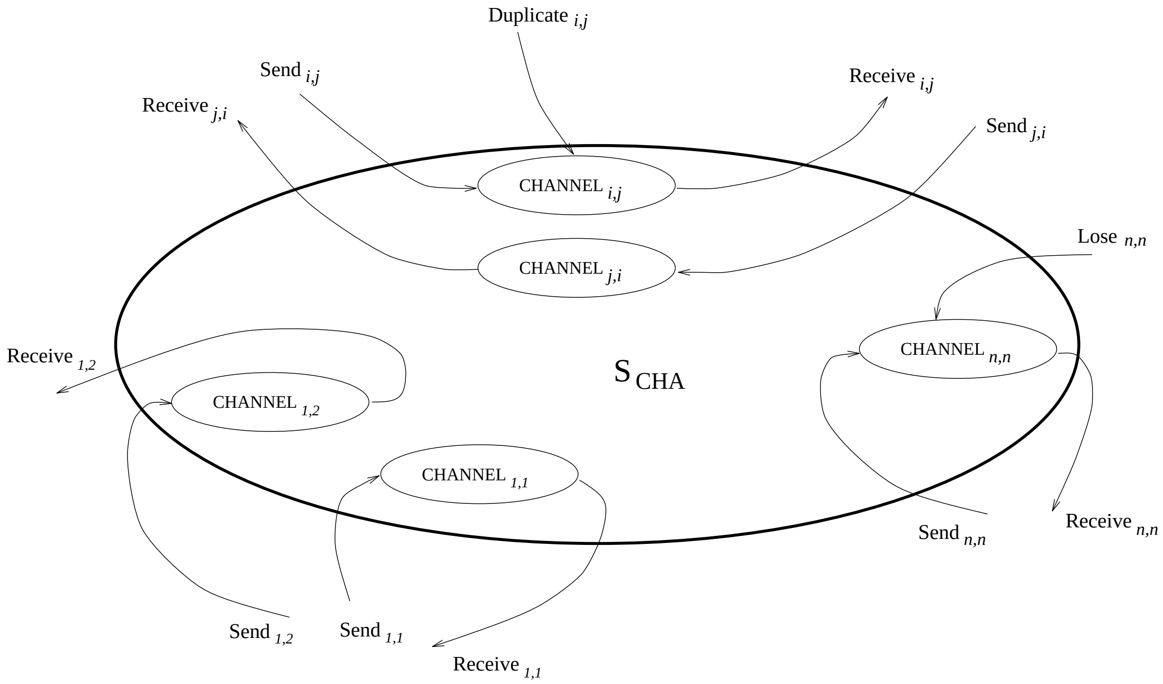

Figure 3-2: The communication system S_{CHA}

Next we provide the definition of "stable" execution fragment for a distributed system exploiting the definitions of stable execution fragment given previously for channels and process automata.

Definition 3.3.3 *Given a distributed system* S*, we say that an execution fragment* α *of* S *is "stable" if:*

1. *for all automata* PROCESS_i *modelling process* i, $i \in S$ *it holds that* $\alpha|\text{PROCESS}_i$ *is a stable execution fragment for process* i.

2. *for all channels* CHANNEL$_{i,j}$ *with* $i, j \in S$ *it holds that* $\alpha|$CHANNEL$_{i,j}$ *is a stable execution fragment for* CHANNEL$_{i,j}$.

Finally we provide the definition of "nice" execution fragment that captures the conditions under which PAXOS satisfies termination.

Definition 3.3.4 *Given a distributed system* S*, we say that an execution fragment* α *of* S *is "nice" if* α *is a stable execution fragment and a majority of the processes are alive in* α.

The above definition requires a majority of processes to be alive. As will be explained in Chapter 6, the property of majorities needed by the PAXOS algorithm is that any two majorities have one element on common. Hence any quorum scheme could be used.

In the rest of the thesis, we will use the word "system" to mean "distributed system".

Chapter 4

The consensus problem

Several variations of the consensus problem have been studied. These variations depends on the model used. In this chapter we provide a formal definition of the consensus problem that we consider.

4.1 Overview

In a distributed system processes need to cooperate and fundamental to such cooperation is the problem of reaching agreement on some data upon which the computation depends. Well known practical examples of agreement problems arise in distributed databases, where data managers need to agree on whether to commit or abort a given transaction, and flight control systems, where the airplane control system and the flight surface control system need to agree on whether to continue or abort a landing in progress.

In the absence of failures, achieving agreement is a trivial task. An exchange of information and a common rule to make a decision is enough. However the problem becomes much more complex in the presence of failures.

Several different but related agreement problems have been considered in the literature. All have in common that processes start the computation with initial values and at the end of the computation each process must reach a decision. The variations mostly concern stronger or weaker requirements that the solution to the problem has to satisfy. The requirement that a solution to the problem has to satisfy are captured by three properties, usually called *agreement, validity* and *termination*.

As an example, the agreement condition may state that no two processes decide on different values, the validity condition may state that if all the initial values are equal then the (unique) decision must be equal to the initial value and the termination condition may state that every process must decide. A weaker agreement condition may require that only non-faulty processes agree on the decision (this weaker condition is necessary, for example, when considering Byzantine failures for which the behavior of a faulty process is unconstrained). A stronger validity condition may state that every decision must be equal to some initial value.

It is clear that the definition of the consensus problem must take into account the distributed setting in which the problem is considered.

About synchrony, several model variations, ranging from the completely asynchronous setting to the completely synchronous one, can be considered. A completely asynchronous model is one with no concept of real time. It is assumed that messages are eventually delivered and processes eventually respond, but it may take arbitrarily long. In a completely synchronous model the computation proceeds in a sequence of steps[1]. At each step processes receive messages sent in the previous step, perform some computation and send messages. Steps are taken at regular intervals of time. Thus in a completely synchronous model, processes act as in a single synchronous computer. Between the two extremes of complete synchrony and complete asynchrony, other models with partial synchrony can be considered. These models assume upper bounds on the message transmission time and on the process response time. These upper bounds may be known or unknown to the processes. Moreover processes have some form of real-time clock to take advantage of the time bounds.

Failures may concern both communication channels and processes. In synchronous and partially synchronous models, timing failures are considered. Communication failures can result in loss of messages. Duplication and reordering of messages may

[1] Usually these steps are called “rounds”. However in this thesis we use the word “round” with a different meaning.

be considered failures, too. Models in which incorrect messages may be delivered are seldom considered since there are many techniques to detect the alteration of a message. The weakest assumption made about process failures is that a faulty process has an unrestricted behavior. Such a failure is called a Byzantine failure. Byzantine failures are often considered with authentication; authentication provides a way to sign messages, so that, even a Byzantine-faulty process cannot send a message with the signature of another process. More restrictive models permit only omission failures, in which a faulty process fails to send some messages. The most restrictive models allow only stopping failures, in which a failed process simply stops and takes no further actions. Some models assume that failed processes can be restarted. Often it is assumed that there is some form of stable storage that is not affected by a stopping failure; a stopped process is restarted with its stable storage in the same state as before the failure and with every other part of its state restored to some initial values. In synchronous and partially synchronous models messages are supposed to be delivered and processes are expected to act within some time bounds. A timing failure is a violation of those time bounds.

Real distributed systems are often partially synchronous systems subject to process and channel failures. Though timely responses can be provided in real distributed systems, the possibility of process and channels failures makes impossible to guarantee that timing assumptions are always satisfied. Thus real distributed systems suffer timing failures, too. The possibility of timing failures in a partially synchronous distributed system means that the system may as well behave like an asynchronous one. Unfortunately, reaching consensus in asynchronous systems, is impossible, unless it is guaranteed that no failures happen [18]. Henceforth, to solve the problem we need to rely on the timing assumptions. Since timing failures are anyway possible, safety properties, that is, agreement and validity conditions, must not depend at all on timing assumptions. However we can rely on the timing assumptions for the termination condition.

4.2 Formal definition

In Section 3 we have described the distributed setting we consider in this thesis. In summary, we consider a partial synchronous system of n processes in a complete network; processes are subject to stop failures and recoveries and have stable storage; channels can lose, duplicate and reorder messages; timing failures are also possible.

Next we give a formal definition of the consensus problem we consider.

For each process i there is an external agent that provides an initial value v by means of an action $\text{Init}(v)_i$[2]. We denote by V the set of possible initial values and, given a particular execution α, we denote by V_α the subset of V consisting of those values actually used as initial values in α, that is, those values provided by $\text{Init}(v)_i$ actions executed in α. A process outputs a decision v by executing an action $\text{Decide}(v)_i$. If a process i executes action $\text{Decide}(v)_i$ more than once then the output value v must be the same.

To solve the consensus problem means to give a distributed algorithm that, for any execution α of the system, satisfies

- **Agreement:** All the $\text{Decide}(v)$ actions in α have the same v.
- **Validity:** For any $\text{Decide}(v)$ action in α, v belongs to V_α.

and, for any admissible execution α, satisfies

- **Termination:** If $\alpha = \beta\gamma$ and γ is a nice execution fragment and for each process i alive in γ an $\text{Init}(v)_i$ action occurs in α when process i is alive, then any process i alive in γ, executes a $\text{Decide}(v)_i$ action in α.

The agreement and termination conditions require, as one can expect, that correct processes "agree" on a particular value. The validity condition is needed to relate the output value to the input values (otherwise a trivial solution, i.e. always output a default value, exists).

[2] We remark that usually it is assumed that for each process i the $\text{Init}(v)_i$ action is executed at most once; however we do not need this assumption.

4.3 Bibliographic notes

PAXOS solves the consensus problem in a partially synchronous distributed system achieving termination when the system executes a nice execution fragment. Allowing timing failures, the partially synchronous system may behave as an asynchronous one. A fundamental theoretical result, proved by Fischer, Lynch and Paterson [18] states that in an asynchronous system there is no consensus algorithm even in the presence of only one stopping failure. Essentially the impossibility result stem from the inherent difficulty of determining whether a process has actually stopped or is only slow.

The PAXOS algorithm was devised by Lamport. In the original paper [29], the PAXOS algorithm is described as the result of discoveries of archaeological studies of an ancient Greek civilization. The PAXOS algorithm is presented by explaining how the parliament of this ancient Greek civilization worked. A proof of correctness is provided in the appendix of that paper. A time-performance analysis is discussed. Many practical optimizations of the algorithm are also discussed. In [29] there is also presented a variation of PAXOS that considers multiple concurrent runs of PAXOS when consensus has to be reached on a sequence of values. We call this variation the MULTIPAXOS algorithm.

MULTIPAXOS can be easily used to implement a data replication algorithm. In [34] a data replication algorithm is provided. It incorporates ideas similar to the ones used in PAXOS.

In the class notes of Principles of Computer Systems [31] taught at MIT, a description of PAXOS is provided using a specification language called SPEC. The presentation in [31] contains the description of how a round of PAXOS is conducted. The leader election problem is not considered. Timing issues are not considered; for example, the problem of starting new rounds is not addressed. A proof of correctness, written also in SPEC, is provided. Our presentation differs from that of [31] in the following aspects: it uses the I/O automata models; it provides all the details of the algorithm; it provides a modular description of the algorithm, including auxiliary parts such as a failure detector module and a leader elector module; along with the proof of correctness, it provides a performance and fault-tolerance analysis. In [32] Lampson provides a brief overview of the PAXOS algorithm together with the key points for proving the correctness of the algorithm.

In [11] three different partially synchronous models are considered. For each of them and for different types of failure an upper bound on the number of failures that can be tolerated is shown, and algorithms that achieve the bounds are given. A model studied in [11] considers a distributed setting similar to the one we consider in this thesis: a partially synchronous distributed system in which upper bounds on the process response time and message delivery time hold eventually; the failures considered are process stop failures (also other models that consider omission failures, Byzantine failures with and without authentication are studied in [11]). The protocol provided in [11], the DLS algorithm for short, needs a linear, in the number of processes, amount of time from the point in which the upper bounds on the process response time and message delivery time start holding. This is similar to the PAXOS performance which requires a linear amount of time to achieve termination when the system executes a nice execution fragment. However the DLS algorithm does not consider process recoveries and it is resilient to a number of process stopping failures which is less or equal to half the number of processes. This can be related to PAXOS by the fact that PAXOS requires a majority of processes alive to reach termination. The PAXOS algorithm is resilient also to channel failures while the DLS algorithm does not consider channel failures.

PAXOS bears some similarities with the standard three-phase commit protocol: both require, in each round, an exchange of 5 messages. However the standard commit protocol requires a reliable leader elector while PAXOS does not. Moreover PAXOS sends information on the value to agree on only in the third message of a round (while the commit protocol sends it in the first message) and because of this, MULTIPAXOS can exchange the first two messages only once for many instances and use only the exchange of the last three messages for each individual consensus problem.

Chapter 5

Failure detector and leader elector

In this chapter we provide a failure detector algorithm and then we use it to implement a leader election algorithm, which in turn will be used in Chapter 6 to implement PAXOS. The failure detector and the leader elector we implement here are both sloppy, meaning that they are guaranteed to give accurate information on the system only in a stable execution. However, this is enough for implementing PAXOS.

5.1 A failure detector

In this section we provide an automaton that detects process failures and recoveries and we prove that the automaton satisfies certain properties that we will need in the rest of the thesis. We do not provide a formal definition of the failure detection problem, however, roughly speaking, the failure detection problem is the problem of checking which processes are alive and which ones are stopped.

Without some knowledge of the passage of time it is not possible to detect failures; thus to implement a failure detector we need to rely on timing assumptions. Figure 5-1 shows a Clock GT automaton, called DETECTOR$(z, c)_i$. In our setting failures and recoveries are modeled by means of actions Stop$_i$ and Recover$_i$. These two actions are input actions of DETECTOR$(z, c)_i$. Moreover DETECTOR$(z, c)_i$ has InformStopped$(j)_i$ and InformAlive$(j)_i$ as output actions which are executed when, respectively, the stopping and the recovering of process j are detected. Automaton DETECTOR$(z, c)_i$

$\textsc{detector}(z, c)_i$

Signature:

Input:	$\text{Receive}(m)_{j,i}$, Stop_i, Recover_i
Internal:	$\text{Check}(j)_i$
Output:	$\text{InformStopped}(j)_i$, $\text{InformAlive}(j)_i$, $\text{Send}(m)_{i,j}$
Time-passage:	$\nu(t)$

State:

$\mathit{Clock} \in \mathbb{R}$	initially arbitrary
$\mathit{Status} \in \{\texttt{alive}, \texttt{stopped}\}$	initially $\texttt{alive}$
$\mathit{Alive} \in 2^{\mathcal{I}}$	initially $\mathcal{I}$
for all $j \in \mathcal{I}$:	
$\mathit{Prevrec}(j) \in R^{\geq 0}$	initially arbitrary
$\mathit{Lastinform}(j) \in R^{\geq 0}$	initially Clock
$\mathit{Lastsend}(j) \in R^{\geq 0}$	initially Clock
$\mathit{Lastcheck}(j) \in R^{\geq 0}$	initially Clock

Actions:

input Stop_i
Eff: $\mathit{Status} := \texttt{stopped}$

input Recover_i
Eff: $\mathit{Status} := \texttt{alive}$

internal $\text{Send}(\text{“Alive”})_{i,j}$
Pre: $\mathit{Status} = \texttt{alive}$
Eff: $\mathit{Lastsend}(j) := \mathit{Clock} + z$

input $\text{Receive}(\text{“Alive”})_{j,i}$
Eff: if $\mathit{Status} = \texttt{alive}$ then
  $\mathit{Prevrec}(j) := \mathit{Clock}$
  if $j \notin \mathit{Alive}$ then
    $\mathit{Alive} := \mathit{Alive} \cup \{j\}$
    $\mathit{Lastcheck}(j) := \mathit{Clock} + c$

internal $\text{Check}(j)_i$
Pre: $\mathit{Status} = \texttt{alive}$
  $j \in \mathit{Alive}$
Eff: $\mathit{Lastcheck}(j) := \mathit{Clock} + c$
  if $\mathit{Clock} > \mathit{Prevrec}(j) + z + d$ then
    $\mathit{Alive} := \mathit{Alive} \setminus \{j\}$

output $\text{InformStopped}(j)_i$
Pre: $\mathit{Status} = \texttt{alive}$
  $j \notin \mathit{Alive}$
Eff: $\mathit{Lastinform}(j) := \mathit{Clock} + \ell$

output $\text{InformAlive}(j)_i$
Pre: $\mathit{Status} = \texttt{alive}$
  $j \in \mathit{Alive}$
Eff: $\mathit{Lastinform}(j) := \mathit{Clock} + \ell$

time-passage $\nu(t)$
Pre: $\mathit{Status} = \texttt{alive}$
Eff: Let t' be such that
    $\forall j, \mathit{Clock} + t' \leq \mathit{Lastinform}(j)$
    $\forall j, \mathit{Clock} + t' \leq \mathit{Lastsend}(j)$
    $\forall j, \mathit{Clock} + t' \leq \mathit{Lastcheck}(j)$
  $\mathit{Clock} := \mathit{Clock} + t'$

Figure 5-1: Automaton $\textsc{detector}$ for process i

works by having each process constantly sending “Alive” messages to each other process and checking that such messages are received from other processes. It sends at least one “Alive” message in an interval of clock time of a fixed length z (i.e., if an “Alive” message is sent at time t then the next one is sent before time $t + z$) and checks for incoming messages at least once in an interval of time of a fixed length c. Let us denote by S_{DET} the system consisting of system S_{CHA} and an automaton DETECTOR$(z, c)_i$ for each process $i \in \mathcal{I}$. Figure 5-2 shows S_{DET}.

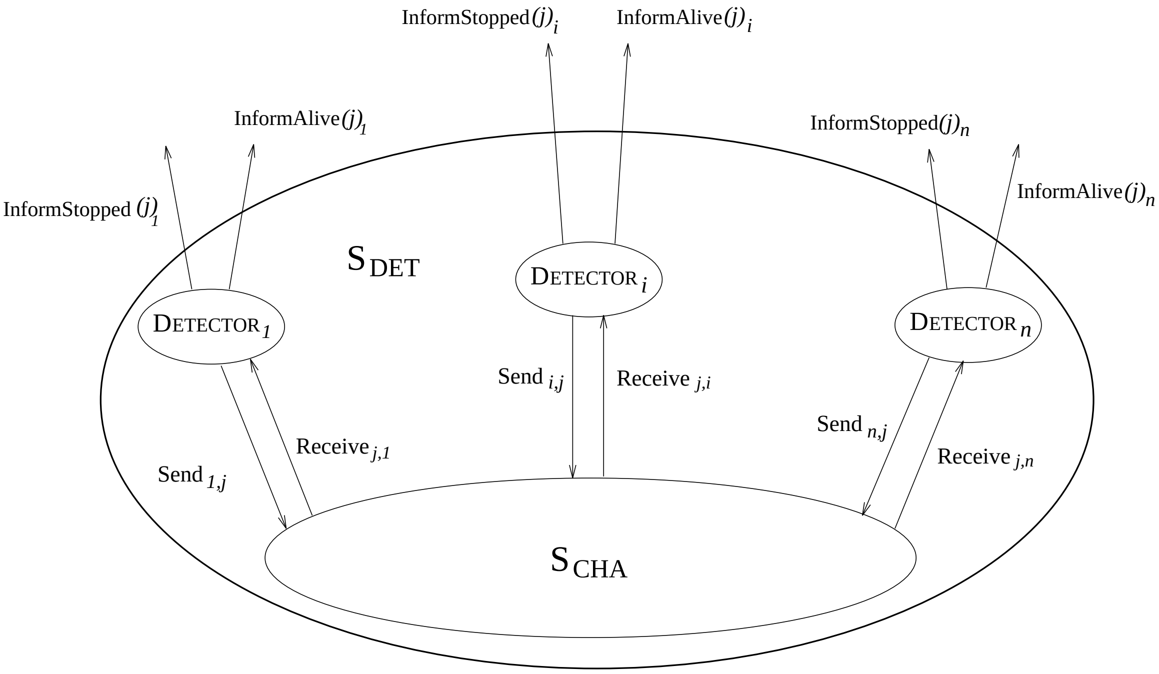

Figure 5-2: The system S_{DET}

The next two lemmas state that if an execution fragment of system S_{DET} is stable then after some time the information provide by the failure detector correctly reflects the status of the processes.

Lemma 5.1.1 *If an execution fragment α of S_{DET}, starting in a reachable state and lasting for more than $z + c + \ell + 2d$ time, is stable and process i is stopped in α, then by time $z + c + \ell + 2d$, for each process j alive in α, an action InformStopped$(i)_j$ is executed and no subsequent InformAlive$(i)_j$ action is executed in α.*

Proof: Let j be any alive process, and let t' be the $Clock_j$ value of process j at the beginning of α. Notice that, since α is stable, at time Δ in α, we have that $Clock_j = t' + \Delta$. Now, notice that $\text{CHANNEL}_{i,j}$ is a subsystem of S_{DET}, that is, S_{DET} is the composition of $\text{CHANNEL}_{i,j}$ and other automata. By Theorem 2.6.10 the projection $\alpha|\text{CHANNEL}_{i,j}$ is an execution fragment of $\text{CHANNEL}_{i,j}$ and thus any property true for $\text{CHANNEL}_{i,j}$ in $\alpha|\text{CHANNEL}_{i,j}$ is true for S_{DET} in α; in particular we can use Lemma 3.2.3. Since α is stable and starts in a reachable state we have that $\alpha|\text{CHANNEL}_{i,j}$ is stable and starts in a reachable state. Thus by Lemma 3.2.3, any message from i to j that is in the channel at the beginning of α is delivered by time d and consequently, since process i is stopped in α, no message from process i is received by process j after time d. We distinguish two possible cases. Let s be the first state of α after which no further messages from i are received by j and let π_s be the action that brings the system into state s. Notice that the time of occurrence of π_s is before or at time d.

CASE 1: Process $i \notin s.Alive_j$. Then, by the code of DETECTOR_i an action InformStopped$(i)_j$ is executed within ℓ time after s. Clearly action InformStopped$(i)_j$ is executed after s and, since the time of occurrence of π_s is $\leq d$ then it is executed before or at time $d + \ell$. Moreover since no messages from i are received after s, no InformAlive$(i)_j$ can happen later on. Thus the lemma is proved in this case.

CASE 2: Process $i \in s.Alive_j$. Let *Prevrec* be the value of $Clock_j$ at the moment when the last "Alive" message from i is received from j. Since no message from process i is received by process j after s and the time of occurrence of π_s is $\leq d$, we have that $Prevrec \leq t' + d$; indeed, as we observed before, at time Δ in α, we have that $Clock_j = t' + \Delta$, for any Δ. Since process i is supposed to send a new "Alive" message within z time from the previous one and the message may take up to d time to be delivered, a new "Alive" message from process i is expected by process j before $Clock_j$ passes the value $Prevrec + z + d$. However, no messages are received when $Clock_j > Prevrec$. By the code of $\text{DETECTOR}(z, c)_j$ an action Check$(i)_j$ occurs after time $Prevrec + z + d$ and before or at time $Prevrec + z + c + d$; indeed, a check action occur at least once in an interval of time of length c. When this action occurs, since $Clock_j > Prevrec + z + d$, it removes process i from the $Alive_j$ set (see code). Thus by time $Prevrec + z + c + d$ process i is not in $Alive_j$. Since $Prevrec \leq t' + d$, we have that process i is not in $Alive_j$ before $Clock_j$ passes $t' + z + c + 2d$. Action InformStopped$(i)_j$ is executed within additional ℓ time, that is before $Clock_j$ passes $t' + z + c + 2d + \ell$. Notice also—and we will need this for the second part of the proof—that this action happens when $Clock_j > t' + z + c + 2d > t' + d$. Thus we have that action InformStopped$(i)_j$ is executed by time $z + c + 2d + \ell$. Since we are considering the case when process i is in $Alive_j$ at time d, action InformStopped$(i)_j$ is executed after time d. This is true for any alive process j. Thus the lemma is proved also in this case.

This concludes the proof of the first part of the lemma. Now, since no messages from i are received by j after time d, that is, no message from i to j is received when $Clock_j > t' + d$ and, by the first part of this proof, InformStopped$(i)_j$ happens when $Clock_j > t' + d$, we have that no InformAlive$(i)_j$ action can occur after InformStopped$(i)_j$ has occurred. This is true for any alive process j. Thus also the second part of the lemma is proved. ■

Lemma 5.1.2 *If an execution fragment α of S_{DET}, starting in a reachable state and lasting for more than $z+d+\ell$ time, is stable and process i is alive in α, then by time $z+d+\ell$, for each process j alive in α, an action InformAlive$(i)_j$ is executed and no subsequent InformStopped$(i)_j$ action is executed in α.*

Proof: Let j be any alive process, and let t' be the value of $Clock_i$ and t'' be the value of $Clock_j$ at the beginning of α. Notice that, since α is stable, at time Δ in α, we have that $Clock_i = t' + \Delta$ and $Clock_j = t'' + \Delta$. Now, notice that $\text{CHANNEL}_{i,j}$ is a subsystem of S_{DET}, that is, S_{DET} can be though of as the composition of $\text{CHANNEL}_{i,j}$ and other automata. By Theorem 2.6.10 $\alpha|\text{CHANNEL}_{i,j}$ is an execution of $\text{CHANNEL}_{i,j}$ and thus any property of $\text{CHANNEL}_{i,j}$ true in $\alpha|\text{CHANNEL}_{i,j}$ is true for S_{DET} in α; in particular we can use Lemma 3.2.3. Since process i is alive in α and α is stable, process i sends an "Alive" message to process j by time z and, by Lemma 3.2.3, such a message is received by process j by time $z + d$. Whence, before $Clock_j$ passes $t'' + z + d$, action

Receive(“Alive”)$_{i,j}$ is executed and thus process i is put into $Alive_j$ (unless it was already there). Once process i is into $Alive_j$, within additional ℓ time, that is before $Clock_j$ passes $t'' + z + d + \ell$, or equivalently, by time $z + d + \ell$, action InformAlive$(i)_j$ is executed. This is true for any process j. This proves the first part of the Lemma.

Let t be the time of occurrence of the first Receive(“Alive”)$_{i,j}$ executed in α; by the first part of this lemma, $t \leq z + d$. Then since α is stable, process i sends at least one “Alive” message in an interval of time z and each message takes at most d to be delivered. Thus in any interval of time $z + d$ process j executes a Receive(“Alive”)$_{i,j}$. This implies that the $Clock_j$ variable of process j never assumes values greater than $Prevrec(i)_j + z + d$, which in turns imply that every Check$(i)_j$ action does not remove process i from $Alive_j$. Notice that process i may be removed from $Alive_j$ before time t. However it is put into $Alive_j$ at time t and it is not removed later on. Thus also the second part of the lemma is proved. ■

The strategy used by DETECTOR$(z, c)_i$ is a straightforward one. For this reason it is very easy to implement. However the failure detector so obtained is not reliable, i.e., it does not give accurate information, in the presence of failures (Stop$_i$, Lose$_{i,j}$, irregular executions). For example, it may consider a process stopped just because the “Alive” message of that process was lost in the channel. Automaton DETECTOR$(z, c)_i$ is guaranteed to provide accurate information on faulty and alive processes only when the system is stable.

In the rest of this thesis we assume that $z = \ell$ and $c = \ell$, that is, we use DETECTOR$(\ell, \ell)_i$. This particular strategy consists of sending an “Alive” message in each interval of ℓ time (i.e., we assume $z = \ell$) and of checking for incoming messages at least once in each interval of ℓ time (i.e., we assume $c = \ell$). In practice the choice of z and c may be different. However for the sake of simplicity we get rid of parameters z and c by assuming $z = \ell$ and $c = \ell$. Lemmas 5.1.3 and 5.1.4 can be restated as follows.

Lemma 5.1.3 *If an execution fragment α of S_{DET}, starting in a reachable state and lasting for more than $3\ell + 2d$ time, is stable and process i is stopped in α, then by* *time $3\ell + 2d$, for each process j alive in α, an action InformStopped$(i)_j$ is executed and no subsequent InformAlive$(i)_j$ action is executed in α.*

Lemma 5.1.4 *If an execution fragment α of S_{DET}, starting in a reachable state and lasting for more than $d + 2\ell$ time, is stable and process i is alive in α, then by time $d + 2\ell$, for each process j alive in α, an action InformAlive$(i)_j$ is executed and no subsequent InformStopped$(i)_j$ action is executed in α.*

5.2 A leader elector

Electing a leader in an asynchronous distributed system is a difficult task. An informal argument that explains this difficulty is that the leader election problem is somewhat similar to the consensus problem, which, in an asynchronous system subject to failures is unsolvable [18], in the sense that to elect a leader all processes must reach consensus on which one is the leader. As for the failure detector, we need to rely on timing assumptions. It is fairly clear how a failure detector can be used to elect a leader. Indeed the failure detector gives information on which processes are alive and which ones are not alive. This information can be used to elect the current leader. We use the DETECTOR$(\ell, \ell)_i$ automaton to check for the set of alive processes. Figure 5-3 shows automaton LEADERELECTOR$_i$ which is an MMT automaton. Remember that we use MMT automata to describe in a simpler way some Clock GT automata. Automaton LEADERELECTOR$_i$ interacts with DETECTOR$(\ell, \ell)_i$ by means of actions InformStopped$(j)_i$, which inform process i that process j has stopped, and InformAlive$(j)_i$, which inform process i that process j has recovered. Each process updates its view of the set of alive processes when these two actions are executed. The process with the biggest identifier in the set of alive processes is declared leader. We denote with S_{LEA} the system consisting of S_{DET} composed with a LEADERELECTOR$_i$ automaton for each process $i \in \mathcal{I}$. Figure 5-4 shows S_{LEA}.

Since DETECTOR$(\ell, \ell)_i$ is not a reliable failure detector, also LEADERELECTOR$_i$ is not reliable. Thus, it is possible that processes have different views of the system so that more than one process considers itself leader, or the process supposed to be the

LEADERELECTOR$_i$

Signature:

Input: InformStopped$(j)_i$, InformAlive$(j)_i$, Stop$_i$, Recover$_i$
Output: Leader$_i$, NotLeader$_i$

State:

$Status \in \{\texttt{alive}, \texttt{stopped}\}$ initially `alive`
$Pool \in 2^{\mathcal{I}}$ initially $\{i\}$

Derived variable:

Leader, defined as max of *Pool*

Actions:

input Stop$_i$
Eff: $Status := \texttt{stopped}$

output Leader$_i$
Pre: $Status = \texttt{alive}$
$i = Leader$
Eff: none

input InformStopped$(j)_i$
Eff: if $Status = \texttt{alive}$ then
$Pool := Pool \setminus \{j\}$

input Recover$_i$
Eff: $Status := \texttt{alive}$

output NotLeader$_i$
Pre: $Status = \texttt{alive}$
$i \neq Leader$
Eff: none

input InformAlive$(j)_i$
Eff: if $Status = \texttt{alive}$
$Pool := Pool \cup \{j\}$

Tasks and bounds:

$\{$Leader$_i$, NotLeader$_i\}$, bounds $[0, \ell]$

Figure 5-3: Automaton LEADERELECTOR for process i

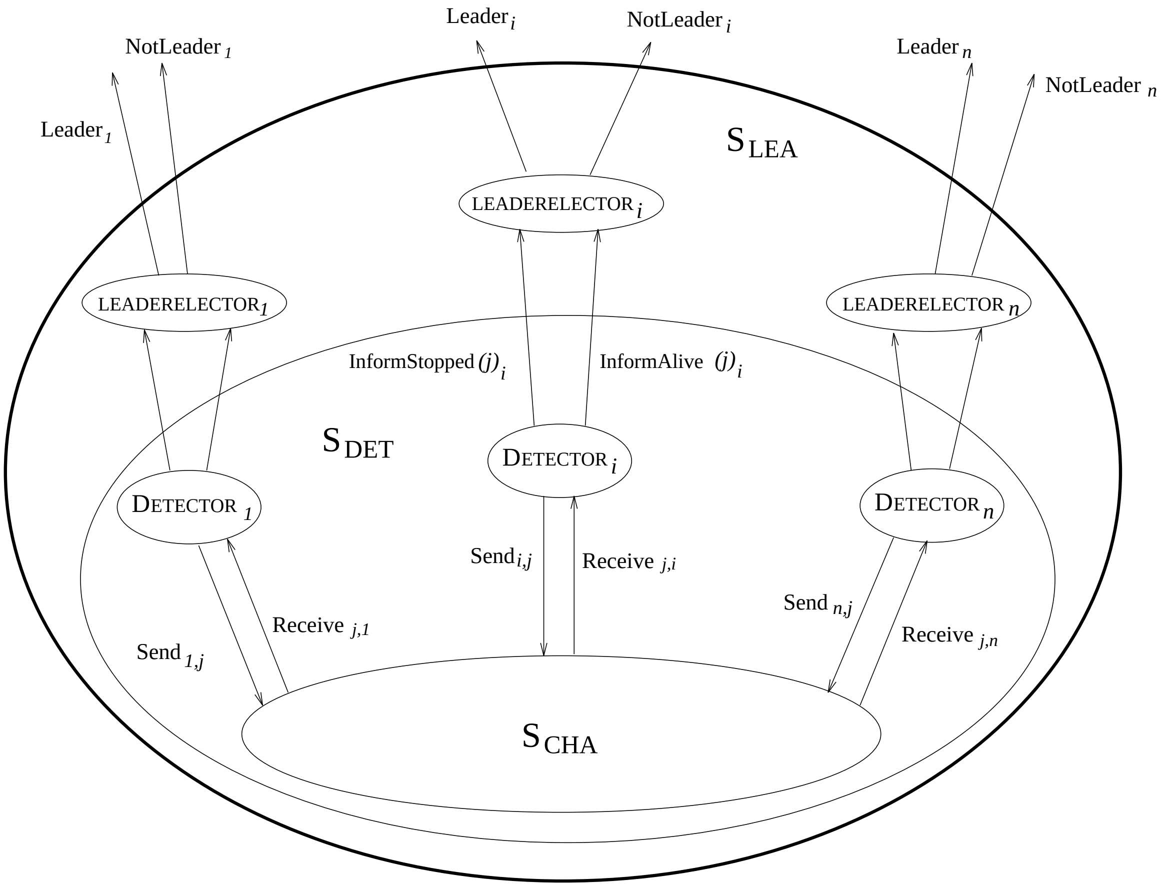

Figure 5-4: The system S_{LEA}

leader is actually stopped. However as the failure detector becomes reliable when the system S_{DET} executes a stable execution fragment (see Lemmas 5.1.3 and 5.1.4), also the leader elector becomes reliable when system S_{LEA} is stable. Notice that when S_{LEA} executes a stable execution fragment, so does S_{DET}.

Formally we consider a process i to be *leader* if $Leader_i = i$. That is, a process i is leader if it consider itself to be the leader. This allows multiple or no leaders and does not require other processes to be aware of the leader or the leaders. The following definitions give a much more precise notion of leader.

Definition 5.2.1 *A state s of system S_{LEA}, is a "unique-leader" state if there exists an alive process i such that for all alive processes j it holds that $s.Leader_j = i$. Process i is the "leader" in state s.*

Definition 5.2.2 *An execution α of system S_{LEA}, is a "unique-leader" execution if* *all the states of α are unique-leader states with the same leader in all the states.*

The next lemma states that in a stable execution fragment, eventually there will be a unique-leader state.

Lemma 5.2.3 *If an execution fragment α of S_{LEA}, starting in a reachable state and lasting for more than $4\ell+2d$, is stable, then by time $4\ell+2d$, there is a state occurrence s such that state s and all the states after s are unique-leader states. Moreover the unique leader is always the process with the biggest identifier among the processes alive in α.*

Proof: First notice that the system S_{LEA} consists of system S_{DET} composed with other automata. Hence by Theorem 2.6.10 we can use any property of S_{DET}. In particular we can use Lemmas 5.1.3 and 5.1.4 and thus we have that by time $3\ell + 2d$ each process has a consistent view of the set of alive and stopped processes. Let i be the leader. Since α is stable and thus also regular, by Lemma 2.5.4, within additional ℓ time, actions Leader$_j$ and NotLeader$_j$ are consistently executed for each process j, including process $j = i$. The fact that i is the the process with the biggest identifier among the processes alive in α follows directly from the code of LEADERELECTOR$_i$. ■

We remark that, for many algorithms that rely on the concept of leader, it is important to provide exactly one leader. For example when the leader election is used to generate a new token in a token ring network, it is important that there is exactly one process (the leader) that generates the new token, because the network gives the right to send messages to the owner of the token and two tokens may result in an interference between two communications. For these algorithms, having two or more leaders jeopardizes the correctness. Hence the sloppy leader elector provided before is not suitable. However for the purpose of this thesis, LEADERELECTOR$_i$ is all we need.

5.3 Bibliographic notes

In an asynchronous system it is impossible to distinguish a very slow process from a stopped one. This is why the consensus problem cannot be solved even in the case where at most one process fails [18]. If a reliable failure detector were provided then the consensus problem would be solvable. This clearly implies that in a completely asynchronous setting no reliable failure detector can be provided. Chandra and Toueg [5] gave a definition of unreliable failure detector, and characterized failure detectors in terms of two properties: completeness, which requires that the failure detector eventually suspect any stopped process, and accuracy, which restricts the mistakes a failure detector can make. No failure detectors are actually implemented in [5]. The failure detector provided in this thesis differs from those classified by Chandra and Toueg in that it provides reliability conditional to the system stabilization. If the system eventually stabilizes then our failure detector can be classified in the class of the eventually perfect failure detectors. However it should be noted that in order to achieve termination we actually do not need that the system become stable forever but only for a sufficiently long time.

Chandra, Hadzilacos and Toueg [4] identified the “weakest” failure detector that can be used to solve the consensus problem.

Failure detectors have practical relevance since it is often important to establish which processes are alive and which ones are stopped. The need for a leader in a distributed computation arises in many practical situations, like, for example, in a token ring network. However in asynchronous systems there is the inherent difficulty of distinguishing a stopped process from a slow one.

Chapter 6

The PAXOS algorithm

PAXOS was devised a very long time ago[1] but its discovery, due to Lamport, is very recent [29].

In this chapter we describe the PAXOS algorithm, provide an implementation using Clock GT automata, prove its correctness and analyze its performance. The performance analysis is given assuming that there are no failures nor recoveries, and a majority of the processes are alive for a sufficiently long time. We remark that when no restrictions are imposed on the possible failures, the algorithm might not terminate.

6.1 Overview

Our description of PAXOS is modular: we have separated various parts of the overall algorithm; each piece copes with a particular aspect of the problem. This approach should make the understanding of the algorithm much easier. The core part of the algorithm is a module that we call BASICPAXOS; this piece incorporates the basic ideas on which the algorithm itself is built. The description of this piece is further subdivided into three components, namely BPLEADER, BPAGENT and BPSUCCESS.

In BASICPAXOS processes try to reach a decision by running what we call a "round". A process starting a round is the leader of that round. BASICPAXOS guar-

[1] The most accurate information dates it back to the beginning of this millennium [29].

antees that, no matter how many leaders start rounds, agreement and validity are not violated. However to have a complete algorithm that satisfies termination when there are no failures for a sufficiently long time, we need to augment BASICPAXOS with another module; we call this module STARTERALG. The functionality of STARTERALG is to make the current leader start a new round if the previous one is not completed within some time bound.

Leaders are elected by using the LEADERELECTOR algorithm provided in Chapter 5. We remark that this is possible because the presence of two or more leaders does not jeopardize agreement or validity; however, to get termination there must be a unique leader.

Thus, our implementation of PAXOS is obtained by composing the following automata: $\text{CHANNEL}_{i,j}$ for the communication between processes, DETECTOR_i and LEADERELECTOR_i for the leader election, BASICPAXOS_i and STARTERALG_i, for every process $i, j \in \mathcal{I}$. The resulting system is called S_{PAX}.

Figure 6-1 shows the automaton at process i. Notice that not all of the actions are drawn in the picture: we have drawn only some of them and we refer to the formal code for all of the actions. Actions Stop_i and Recover_i are input actions of all the automata. The S_{PAX} automaton at process i interacts with automata at other processes by sending messages over the channels. Channels are not drawn in the picture.

Figure 6-2 shows the messages exchanged by processes i and j. The automata that send and receive these messages are shown in the picture. We remark that channels and actions interacting with channels are not drawn, as well as other actions for the interaction with other automata.

It is worth to remark that some pieces of the algorithm do need to be able to measure the passage of the time (DETECTOR_i, STARTERALG_i and BPSUCCESS_i) while others do not.

We will prove (Theorems 6.2.15 and 6.2.18) that the system S_{PAX} solves the consensus problem ensuring partial correctness—any output is guaranteed to be correct, that is, agreement and validity are satisfied—and (Theorem 6.4.2) that S_{PAX} guar-

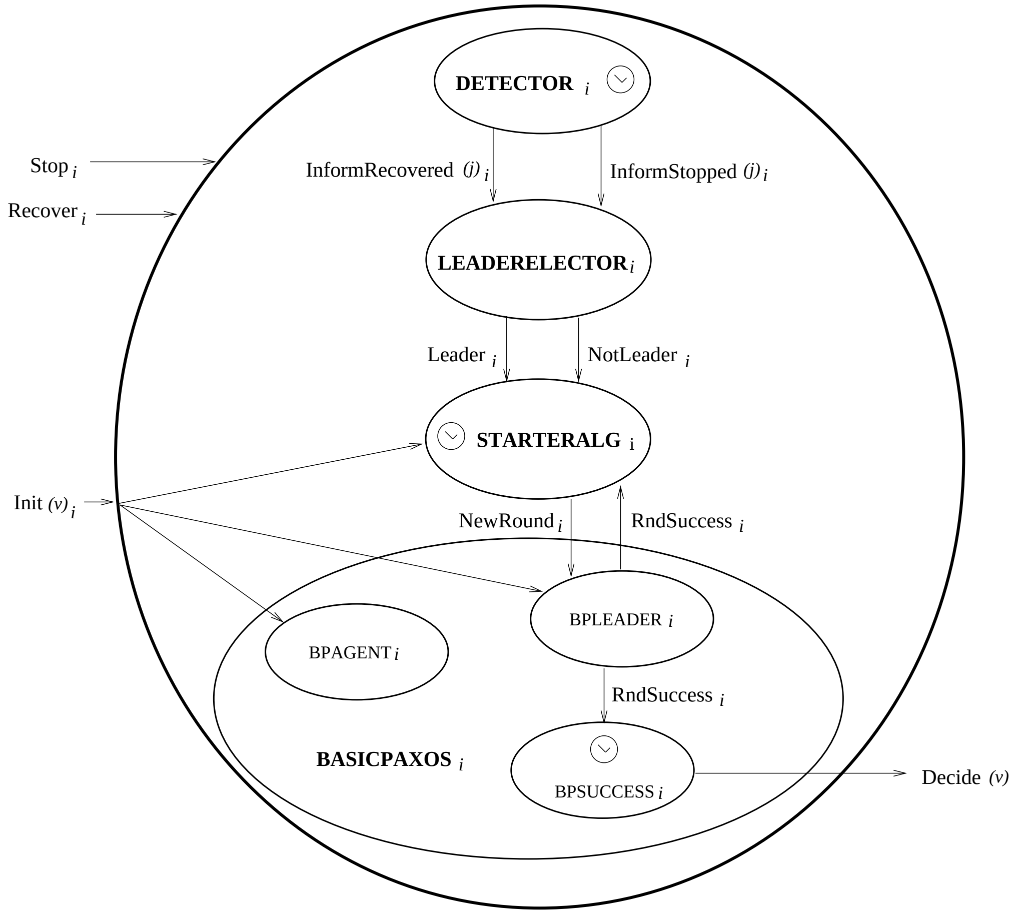

Figure 6-1: PAXOS: process i.

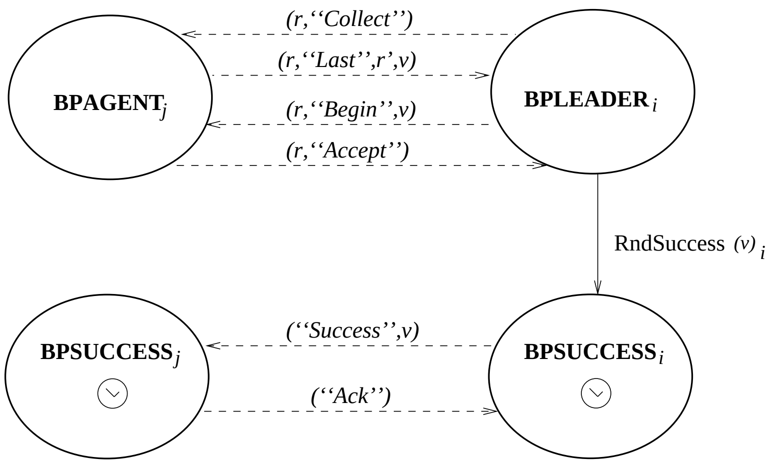

Figure 6-2: BASICPAXOS: Messages.

antees also termination when the system executes a nice execution fragment, that is, without failures and recoveries and with at least a majority of the processes remaining alive.

Roadmap for the rest of the chapter. In Section 6.2 we provide automaton BASICPAXOS. This automaton is responsible for carrying out a round in response to an external request. We prove that any round satisfies agreement and validity and we provide a performance analysis for a successful round. Then in Section 6.3 we provide automaton STARTERALG which takes care of the problem of starting new rounds. In Section 6.4 we prove that the entire system S_{PAX} is correct and provide a performance analysis. In Section 6.5 we provide some comments about the number of messages used by the algorithm. Finally Section 6.6 contains some concluding remarks.

6.2 Automaton BASICPAXOS

In this section we present the automaton BASICPAXOS which is the core part of the PAXOS algorithm. We begin by providing an overview of how automaton BASICPAXOS works, then we provide the automaton code along with a detailed description and finally we prove that it satisfies agreement and validity.

6.2.1 Overview

The basic idea is to have processes propose values until one of them is accepted by a majority of the processes; that value is the final output value. Any process may propose a value by initiating a *round* for that value. The process initiating a round is said to be the *leader* of that round while all processes, including the leader itself, are said to be *agents* for that round. Informally, the steps for a round are the following.

1. To initiate a round, the leader sends a "Collect" message to all agents[2] announcing that it wants to start a new round and at the same time asking for

[2]Thus it sends a message also to itself. This helps in that we do not have to specify different behaviors for a process according to the fact that it is both leader and agent or just an agent. We just need to specify the leader behavior and the agent behavior.

information about previous rounds in which agents may have been involved.

2. An agent that receives a message sent in step 1 from the leader of the round, responds with a “Last” message giving its own information about rounds previously conducted. With this, the agent makes a kind of commitment for this particular round that may prevent it from accepting (in step 4) the value proposed in some other round. If the agent is already committed for a round with a bigger round number then it informs the leader of its commitment with an “OldRound” message.

3. Once the leader has gathered information about previous rounds from a majority of agents, it decides, according to some rules, the value to propose for its round and sends to all agents a “Begin” message announcing the value and asking them to accept it. In order for the leader to be able to choose a value for the round it is necessary that initial values be provided. If no initial value is provided, the leader must wait for an initial value before proceeding with step 3. The set of processes from which the leader gathers information is called the *info-quorum* of the round.

4. An agent that receives a message from the leader of the round sent in step 3, responds with an “Accept” message by accepting the value proposed in the current round, unless it is committed for a later round and thus must reject the value proposed in the current round. In the latter case the agent sends an “OldRound” message to the leader indicating the round for which it is committed.

5. If the leader gets “Accept” messages from a majority of agents, then the leader sets its own output value to the value proposed in the round. At this point the round is successful. The set of agents that accept the value proposed by the leader is called the *accepting-quorum*.

Since a successful round implies that the leader of the round reached a decision, after a successful round the leader still needs to do something, namely to broadcast the reached decision. Thus, once the leader has made a decision it broadcasts a

"Success" message announcing the value for which it has decided. An agent that receives a "Success" message from the leader makes its decision choosing the value of the successful round. We use also an "Ack" message sent from the agent to the leader, so that the leader can make sure that everyone knows the outcome.

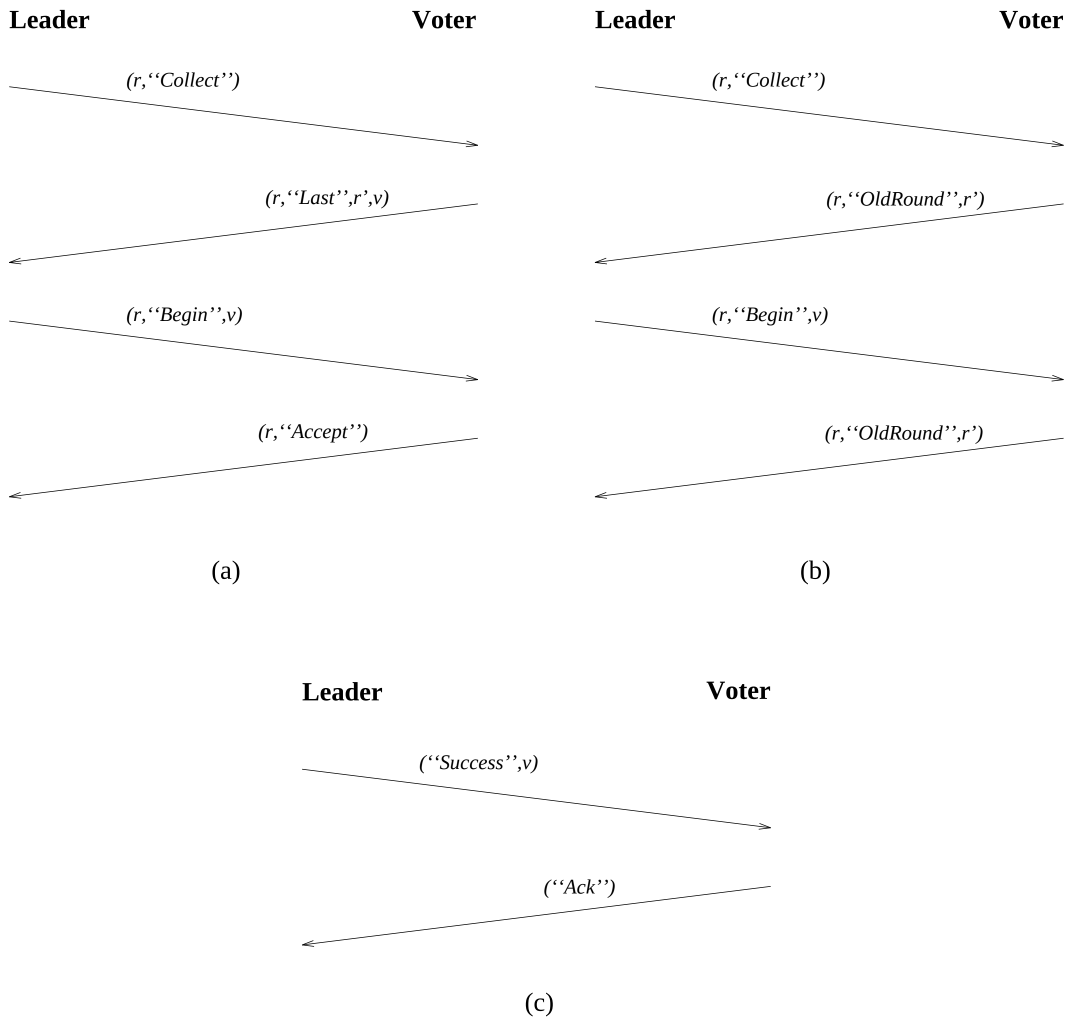

Figure 6-3: Exchange of messages

Figure 6-3 shows: (a) the steps of a successful round r; (b) the responses from an agent that informs the leader that an higher numbered round r' has been already initiated; (c) the broadcast of a decision. The parameters used in the messages will be explained later. Section 6.2.2 contains a description of the messages.

Since different rounds may be carried out concurrently (several processes may concurrently initiate rounds), we need to distinguish them. Every round has a unique identifier. Next we formally define these round identifiers. A *round number* is a pair (x, i) where x is a nonnegative integer and i is a process identifier. The set of round numbers is denoted by $\mathcal{R}$. A total order on elements of $\mathcal{R}$ is defined by $(x, i) < (y, j)$ iff $x < y$ or, $x = y$ and $i < j$.

Definition 6.2.1 *Round r "precedes" round r' if $r < r'$.*

If round r precedes round r' then we also say that r is a *previous* round, with respect to round r'. We remark that the ordering of rounds is not related to the actual time the rounds are conducted. It is possible that a round r' is started at some point in time and a previous round r, that is, one with $r < r'$, is started later on.

For each process i, we define a "$+_i$" operation that given a round number (x, j) and an integer y, returns the round number $(x, j) +_i y = (x + y, i)$.

Every round in the algorithm is tagged with a unique round number. Every message sent by the leader or by an agent for a round (with round number) $r \in \mathcal{R}$, carries the round number r so that no confusion among messages belonging to different rounds is possible.

However the most important issue is about the values that leaders propose for their rounds. Indeed, since the value of a successful round is the output value of some processes, we must guarantee that the values of successful rounds are all equal in order to satisfy the agreement condition of the consensus problem. This is the tricky part of the algorithm and basically all the difficulties derive from solving this problem. Consistency is guaranteed by choosing the values of new rounds exploiting the information about previous rounds from at least a majority of the agents so that, for any two rounds, there is at least one process that participated in both rounds.

In more detail, the leader of a round chooses the value for the round in the following way. In step 1, the leader asks for information and in step 2 an agent responds with the number of the latest round in which it accepted the value and with the accepted value or with round number $(0, j)$ and `nil` if the agent has not yet accepted a value. Once the leader gets such information from a majority of the agents (which is the info-quorum of the round), it chooses the value for its round to be equal to the value of the latest round among all those it has heard from the agents in the info-quorum or equal to its initial value if all agents in the info-quorum were not involved in any previous round. Moreover, in order to keep consistency, if an agent tells the leader of a round r that the last round in which it accepted a value is round r', $r' < r$, then implicitly the agent commits itself not to accept any value proposed in any other round r'', $r' < r'' < r$.

Given the above setting, if r' is the round from which the leader of round r gets the value for its round, then, when a value for round r has been chosen, any round r'', $r' < r'' < r$, cannot be successful; indeed at least a majority of the processes are committed for round r, which implies that at least a majority of the processes are rejecting round r''. This, along with the fact that info-quorums and accepting-quorums are majorities, implies that if a round r is successful, then any round with a bigger round number $\tilde{r} > r$ is for the same value. Indeed the information sent by processes in the info-quorum of round $\tilde{r}$ is used to choose the value for the round, but since info-quorums and accepting-quorums share at least one process, at least one of the processes in the info-quorum of round r' is also in the accepting-quorum of round r. Indeed, since the round is successful, the accepting-quorum is a majority. This implies that the value of any round $\tilde{r} > r$ must be equal to the value of round r, which, in turn, implies agreement.

We remark that instead of majorities for info-quorums and accepting-quorums, any quorum system can be used. Indeed the only property that is required is that there be a process in the intersection of any info-quorum with any accepting-quorum.

Example. Figure 6-4 shows how the value of a round is chosen. In this example we have a network of 5 processes, A, B, C, D, E (where the ordering is the alphabetical one) and v_A, v_B denote the initial values of A and B. At some point process B is the leader and starts round $(1, B)$. It receives information from A, B, E (the set $\{A, B, E\}$ is the info-quorum of this round). Since none of them has been involved in a previous round, process B is free to choose its initial value v_B as the value of the round. However it receives acceptance only from B, C (the set $\{B, C\}$ is the accepting-quorum for this round). Later, process A becomes the leader and starts round $(2, A)$. The info-quorum for this round is $\{A, D, E\}$.

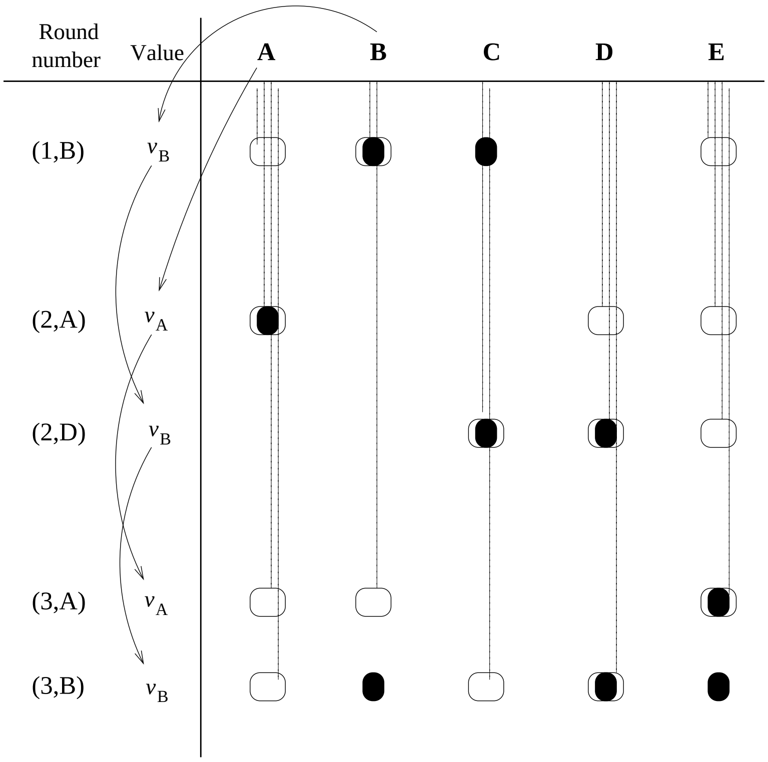

Figure 6-4: Choosing the values of rounds. Empty boxes denote that the process is in the info-quorum, and black boxes denote acceptance. Dotted lines indicate commitments.

Since none of this processes has accepted a value in a previous round, A is free to choose its initial value for its round. For round $(2, D)$ the info-quorum is $\{C, D, E\}$. This time in the quorum there is process C that has accepted a value in round $(1, B)$ so the value of this round must be the same of that of round $(1, B)$. For round $(3, A)$ the info-quorum is $\{A, B, E\}$ and since A has accepted the value of round $(2, A)$ then the value of round $(2, A)$ is chosen for round $(3, A)$. For round $(3, B)$ the info-quorum is $\{A, C, D\}$. In this case there are three processes that accepted values in previous rounds: process A that has accepted the value of round $(2, A)$ and processes C, D, that have accepted the value of round $(2, D)$. Since round $(2, D)$ is the higher round number, the value for round $(3, B)$ is taken from round $(2, D)$. Round $(3, B)$ is successful.

To end up with a decision value, rounds must be started until at least one is successful. The basic consensus module BASICPAXOS guarantees that a new round does not violate agreement or validity, that is, the value of a new round is chosen in such a way that if the round is successful, it does not violate agreement and validity. However, it is necessary to make BASICPAXOS start rounds until one is successful. We deal with this problem in Section 6.3.

6.2.2 The code

In order to describe automaton BASICPAXOS$_i$ for process i we provide three automata. One is called BPLEADER$_i$ and models the "leader" behavior of the process; another one is called BPAGENT$_i$ and models the "agent" behavior of the process; the third one is called BPSUCCESS$_i$ and it simply takes care of broadcasting a reached decision. Automaton BASICPAXOS$_i$ is the composition of BPLEADER$_i$, BPAGENT$_i$ and BPSUCCESS$_i$.

Figures 6-5 and 6-6 show the code for BPLEADER$_i$, while Figure 6-7 shows the code for BPAGENT$_i$. We remark that these code fragments are written using the MMTA model. Remember that we use MMTA to describe in a simpler way Clock GT automata. In section 2.3 we have described a standard technique to transform any MMTA into a Clock GTA. Figures 6-8 and 6-9 show automaton BPSUCCESS$_i$. The purpose of this automaton is simply to broadcast the decision once it has been reached by the leader of a round. Figures 6-2 and 6-3 describe the exchange of messages used in a round.

It is worth noticing that the code fragments are "tuned" to work efficiently when there are no failures. Indeed messages for a given round are sent only once, that is, no attempt is made to try to cope with losses of messages and responses are expected to be received within given time bounds. Other strategies to try to conduct a successful round even in the presence of some failures could be used. For example, messages could be sent more than once to cope with the loss of some messages or a leader could wait more than the minimum required time before abandoning the current round and starting a new one—this is actually dealt with in Section 6.3. We have chosen to send only one message for each step of the round: if the execution is nice, one message is enough to conduct a successful round. Once a decision has been made, there is nothing to do but try to send it to others. Thus once the decision has been made

BPLEADER$_i$

Signature:

Input: Receive$(m)_{j,i}$, $m \in \{$"Last", "Accept","OldRound"$\}$
Init$(v)_i$, NewRound$_i$, Stop$_i$, Recover$_i$, Leader$_i$, NotLeader$_i$
Internal: Collect$_i$, BeginCast$_i$,
GatherLast$(m)_i$, m is a "Last" message
GatherAccept$(m)_i$, m is a "Accept" message
GatherOldRound$(m)_i$ m is a "OldRound" message
Output: Send$(m)_{i,j}$, $m \in \{$"Collect", "Begin"$\}$
Gathered$(v)_i$, Continue$_i$, RndSuccess$(v)_i$

State:

Status $\in$ {`alive`,`stopped`}	initially `alive`	*CurRnd* $\in \mathcal{R}$	initially $(0, i)$
IamLeader, a boolean	initially `false`	*HighestRnd* $\in \mathcal{R}$	initially $(0, i)$
Mode $\in$ {`collect`,`gatherlast`, `wait`, `begincast`, `gatheraccept`, `decided`,`done`}	initially `done`	*Value* $\in V \cup$ {`nil`}	initially `nil`
		ValFrom $\in \mathcal{R}$	initially $(0, i)$
		InfoQuo $\in 2^{\mathcal{I}}$	initially $\{\}$
		AcceptQuo $\in 2^{\mathcal{I}}$	initially $\{\}$
InitValue $\in V \cup$ `nil`	initially `nil`	*InMsgs*, multiset of msgs	initially $\{\}$
Decision $\in V \cup$ {`nil`}	initially `nil`	*OutMsgs*, multiset of msgs	initially $\{\}$

Derived Variable:

LeaderAlive, a boolean, `true` iff *Status* = `alive` and *IamLeader* = `true`

Actions:

input Stop$_i$
Eff: *Status* := `stopped`

input Leader$_i$
Eff: if *Status* = `alive` then
IamLeader := `true`

output Send$(m)_{i,j}$
Pre: *Status* = `alive`
$m_{i,j} \in$ *OutMsgs*
Eff: remove $m_{i,j}$ from *OutMsgs*

input Recover$_i$
Eff: *Status* := `alive`

input NotLeader$_i$
Eff: if *Status* = `alive` then
IamLeader := `false`

input Receive$(m)_{j,i}$
Eff: if *Status* = `alive` then
add $m_{j,i}$ to *InMsgs*

input Init$(v)_i$
Eff: if *Status* = `alive` then
InitValue := v

Figure 6-5: Automaton BPLEADER for process i (part 1)

Actions:

input NewRound_i
Eff: if $\mathit{LeaderAlive} = \texttt{true}$ then
$\mathit{CurRnd} := \mathit{HighestRnd} +_i 1$
$\mathit{HighestRnd} := \mathit{CurRnd}$
$\mathit{Mode} := \texttt{collect}$

output Collect_i
Pre: $\mathit{LeaderAlive} = \texttt{true}$
$\mathit{Mode} = \texttt{collect}$
Eff: $\mathit{ValFrom} := (0, i)$
$\mathit{InfoQuo} := \{\}$
$\mathit{AcceptQuo} := \{\}$
$\forall j$ put $\langle \mathit{CurRnd}, \text{“Collect”}\rangle_{i,j}$ in $\mathit{OutMsgs}$
$\mathit{Mode} := \texttt{gatherlast}$

internal $\text{GatherLast}(m)_i$
Pre: $\mathit{LeaderAlive} = \texttt{true}$
$\mathit{Mode} = \texttt{gatherlast}$
$m = \langle r, \text{“Last”}, r', v\rangle_{j,i} \in \mathit{InMsgs}$
$\mathit{CurRnd} = r$
Eff: remove m from InMsgs
$\mathit{InfoQuo} := \mathit{InfoQuo} \cup \{j\}$
if $\mathit{ValFrom} < r'$ and $v \neq \texttt{nil}$ then
$\mathit{Value} := v$
$\mathit{ValFrom} := r'$
if $|\mathit{InfoQuo}| > n/2$ then
$\mathit{Mode} := \texttt{gathered}$

output $\text{Gathered}(\mathit{Value})$
Pre: $\mathit{LeaderAlive} = \texttt{true}$
$\mathit{Mode} = \texttt{gathered}$
Eff: if $\mathit{Value} = \texttt{nil}$ and $\mathit{InitValue} \neq \texttt{nil}$ then
$\mathit{Value} := \mathit{InitValue}$
if $\mathit{Value} \neq \texttt{nil}$ then
$\mathit{Mode} := \texttt{begincast}$
else
$\mathit{Mode} := \texttt{wait}$

internal Continue_i
Pre: $\mathit{LeaderAlive} = \texttt{true}$
$\mathit{Mode} = \texttt{wait}$
$\mathit{Value} = \texttt{nil}$
$\mathit{InitValue} \neq \texttt{nil}$
Eff: $\mathit{Value} := \mathit{InitValue}$
$\mathit{Mode} := \texttt{begincast}$

internal BeginCast_i
Pre: $\mathit{LeaderAlive} = \texttt{true}$
$\mathit{Mode} = \texttt{begincast}$
Eff: $\forall j$ put $\langle \mathit{CurRnd}, \text{“Begin”}, \mathit{Value}\rangle_{i,j}$ in $\mathit{OutMsgs}$
$\mathit{Mode} := \texttt{gatheraccept}$

internal $\text{GatherAccept}(m)_i$
Pre: $\mathit{LeaderAlive} = \texttt{true}$
$\mathit{Mode} = \texttt{gatheraccept}$
$m = \langle r, \text{“Accept”}\rangle_{j,i} \in \mathit{InMsgs}$
$\mathit{CurRnd} = r$
Eff: remove m from InMsgs
$\mathit{AcceptQuo} := \mathit{AcceptQuo} \cup \{j\}$
if $|\mathit{AcceptQuo}| > n/2$ then
$\mathit{Decision} := \mathit{Value}$
$\mathit{Mode} := \texttt{decided}$

output $\text{RndSuccess}(\mathit{Decision})_i$
Pre: $\mathit{LeaderAlive} = \texttt{true}$
$\mathit{Mode} = \texttt{decided}$
Eff: $\mathit{Mode} := \texttt{done}$

internal $\text{GatherOldRound}(m)_i$
Pre: $\mathit{Status} = \texttt{alive}$
$m = \langle r, \text{“OldRound”}, r'\rangle_{j,i} \in \mathit{InMsgs}$
$\mathit{HighestRnd} < r'$
Eff: remove m from InMsgs
$\mathit{HighestRnd} := r'$

Tasks and bounds:

$\{\text{Collect}_i$, $\text{Gathered}(v)_i$, Continue_i, BeginCast_i, $\text{RndSuccess}(v)_i\}$, bounds $[0, \ell]$
$\{\text{GatherLast}(m)_i$, $m \in \mathit{InMsgs}$, m is a “Last” message$\}$, bounds $[0, \ell]$
$\{\text{GatherAccept}(m)_i$, $m \in \mathit{InMsgs}$, m is a “Accept” message$\}$, bounds $[0, \ell]$
$\{\text{GatherOldRound}(m)_i$, $m \in \mathit{InMsgs}$, m is a “OldRound” message$\}$, bounds $[0, \ell]$
$\{\text{Send}(m)_{i,j}$, $m_{i,j} \in \mathit{OutMsgs}\}$, bounds $[0, \ell]$

Figure 6-6: Automaton BPLEADER for process i (part 2)

BPAGENT$_i$

Signature:

Input: Receive$(m)_{j,i}$, $m \in \{$"Collect", "Begin"$\}$
Init$(v)_i$, Stop$_i$, Recover$_i$
Internal: LastAccept$(m)_i$, m is a "Collect" message
Accept$(m)_i$, m is a "Begin" message
Output: Send$(m)_{i,j}$, $m \in \{$"Last", "Accept", "OldRound"$\}$

State:

$\mathit{Status} \in \{\texttt{alive,stopped}\}$	initially `alive`	$\mathit{Commit} \in \mathcal{R}$	initially $(0, i)$
$\mathit{LastR} \in \mathcal{R}$	initially $(0, i)$	*InMsgs*, multiset of msgs	initially $\{\}$
$\mathit{LastV} \in V \cup \{\texttt{nil}\}$	initially `nil`	*OutMsgs*, multiset of msgs	initially $\{\}$

Actions:

input Stop$_i$
Eff: $\mathit{Status} := \texttt{stopped}$

output Send$(m)_{i,j}$
Pre: $\mathit{Status} = \texttt{alive}$
$m \in \mathit{OutMsgs}$
Eff: remove $m_{i,j}$ from *OutMsgs*

internal LastAccept$(m)_i$
Pre: $\mathit{Status} = \texttt{alive}$
$m = \langle r, \text{"Collect"}\rangle_{j,i} \in \mathit{InMsgs}$
Eff: remove m from *InMsgs*
if $r \geq \mathit{Commit}$ then
$\mathit{Commit} := r$
put $\langle r, \text{"Last"}, \mathit{LastR}, \mathit{LastV}\rangle_{i,j}$
in *OutMsgs*
else
put $\langle r, \text{"OldRound"}, \mathit{Commit}\rangle_{i,j}$
in *OutMsgs*

input Recover$_i$
Eff: $\mathit{Status} := \texttt{alive}$

input Receive$(m)_{j,i}$
Eff: if $\mathit{Status} = \texttt{alive}$ then
add $m_{j,i}$ to *InMsgs*

internal Accept$(m)_i$
Pre: $\mathit{Status} = \texttt{alive}$
$m = \langle r, \text{"Begin"}, v\rangle_{j,i} \in \mathit{InMsgs}$
Eff: remove m from *InMsgs*
if $r \geq \mathit{Commit}$ then
put $\langle r, \text{"Accept"}\rangle_{i,j}$ in *InMsgs*
$\mathit{LastR} := r$, $\mathit{LastV} := v$
else
put $\langle r, \text{"OldRound"}, \mathit{Commit}\rangle_{i,j}$
in *OutMsgs*

input Init$(v)_i$
Eff: if $\mathit{Status} = \texttt{alive}$ then
if $\mathit{LastV} = \texttt{nil}$ then $\mathit{LastV} := v$

Tasks and bounds:

$\{$LastAccept$(m)_i$, $m \in \mathit{InMsgs}$, m is a "Collect" message$\}$, bounds $[0, \ell]$
$\{$Accept$(m)_i$, $m \in \mathit{InMsgs}$, m is a "Begin" message$\}$, bounds $[0, \ell]$
$\{$Send$(m)_{i,j}$, $m_{i,j} \in \mathit{OutMsgs}$ $\}$, bounds $[0, \ell]$

Figure 6-7: Automaton BPAGENT for process i

BPSUCCESS$_i$

Signature:

Input: Receive$(m)_{j,i}$, $m \in \{$"Ack", "Success"$\}$
Stop$_i$, Recover$_i$, Leader$_i$, NotLeader$_i$, RndSuccess$(v)_i$
Internal: SendSuccess$_i$, Check$_i$
Output: Send$(m)_{i,j}$, $m \in \{$"Ack", "Success"$\}$
Decide$(v)_i$
Time-passage: $\nu(t)$

State:

$Clock \in \mathbb{R}$	initially arbitrary
$Status \in \{\texttt{alive}, \texttt{stopped}\}$	initially `alive`
$IamLeader$, a boolean	initially `false`
$Decision \in V \cup \{\texttt{nil}\}$	initially `nil`
$Prevsend \in \mathbb{R} \cup \{\texttt{nil}\}$	initially `nil`
$LastCheck \in \mathbb{R} \cup \{\infty\}$	initially ∞
$LastSS \in \mathbb{R} \cup \{\infty\}$	initially ∞
For each $j \in \mathcal{I}$	
$Acked(j)$, a boolean	initially `false`
$LastSendAck(j) \in \mathbb{R} \cup \{\infty\}$	initially ∞
$LastSendSuc(j) \in \mathbb{R} \cup \{\infty\}$	initially ∞
$OutAckMsgs(j)$, set of msgs	initially $\{\}$
$OutSucMsgs(j)$, set of msgs	initially $\{\}$

Actions:

input Stop$_i$
Eff: $Status :=$ `stopped`

input Leader$_i$
Eff: if $Status =$ `alive` and
$IamLeader =$ `false` then
$IamLeader :=$ `true`
if $Decision \neq$ `nil` then
$LastSS := clock + \ell$
$PrevSend :=$ `nil`

output Send$(m)_{i,j}$
Pre: $Status =$ `alive`
$m_{i,j} \in OutAckMsgs(j)$
Eff: $OutAckMsgs(j) := \{\}$
$LastSendAck(j) := \infty$

output Send$(m)_{i,j}$
Pre: $Status =$ `alive`
$m_{i,j} \in OutSucMsgs(j)$
Eff: $OutSucMsgs(j) := \{\}$
$LastSendSuc(j) := \infty$

input Recover$_i$
Eff: $Status :=$ `alive`

input NotLeader$_i$
Eff: if $Status =$ `alive` then
$IamLeader :=$ `false`
$LastSS := \infty$
$LastCheck := \infty$
For each $j \in \mathcal{I}$
$LastSendSuc(j) := \infty$

input Receive$(\langle$"Ack"$\rangle)_{j,i}$
Eff: if $Status =$ `alive` then
$Acked(j) :=$ `true`

input Receive$(\langle$"Success"$,v\rangle)_{j,i}$
Eff: if $Status =$ `alive` then
$Decision := v$
put $\langle$"Ack"$\rangle_{i,j}$ into $OutAckMsgs(j)$
$LastSendAck(j) := Clock + \ell$

Figure 6-8: Automaton BPSUCCESS for process i (part 1)

input RndSuccess$(v)_i$
  Eff: if *Status* = `alive` then
        $Decision := v$
        if $IamLeader =$ `true` then
          $LastSS := Clock + \ell$
          $PrevSend :=$ `nil`

internal SendSuccess$_i$
  Pre: $Status =$ `alive`
        $IamLeader =$ `true`
        $Decision \neq$ `nil`
        $PrevSend =$ `nil`
        $\exists j \neq i$, $Acked(j) =$ `false`
  Eff: $\forall j \neq i$ such that $Acked(j) =$ `false`
        put $\langle$"Success",$Decision\rangle_{i,j}$
          in $OutSucMsgs(j)$
        $LastSendSuc(j) := Clock + \ell$
        $PrevSend := Clock$
        $LastCheck := Clock + (2\ell + 2d) + \ell$
        $LastSS := \infty$

internal Check$_i$
  Pre: $Status =$ `alive`
        $PrevSend \neq$ `nil`
        $Clock > PrevSend + (2\ell + 2d)$
  Eff: $PrevSend :=$ `nil`
        $LastSS := Clock + \ell$
        $LastCheck := \infty$

output Decide$(v)_i$
  Pre: $Status =$ `alive`
        $Decision \neq$ `nil`
        $Decision = v$
  Eff: none

time-passage $\nu(t)$
  Pre: $Status =$ `alive`
  Eff: Let t' be such that
          $Clock{+}t' \leq LastCheck$
          $Clock{+}t' \leq LastSS$
          and for each $j \in \mathcal{I}$
            $Clock{+}t' \leq LastSendAck(j)$
            $Clock{+}t' \leq LastSendSuc(j)$
        $Clock := Clock{+}t'$

Figure 6-9: Automaton BPSUCCESS for process i (part 2)

by the leader, the leader repeatedly sends the decision to the agents until it gets an acknowledgment. We remark that also in this case, in practice, it is important to choose appropriate time-outs for the re-sending of a message; in our implementation we have chosen to wait the minimum amount of time required by an agent to respond to a message from the leader; if the execution is stable this is enough to ensure that only one message announcing the decision is sent to each agent.

We remark that there is some redundancy that derives from having separate automata for the leader behavior and for the broadcasting of the decision. For example, both automata BPLEADER$_i$ and BPSUCCESS$_i$ need to be aware of the decision, thus both have a *Decision* variable (the *Decision* variable of BPSUCCESS$_i$ is updated when action RndSuccess$_i$ is executed by BPLEADER$_i$ after the *Decision* variable of BPLEADER$_i$ is set). Having only one automaton would have eliminated the need of such a duplication. However we preferred to separate BPLEADER$_i$ and BPSUCCESS$_i$ because they accomplish different tasks.

In addition to the code fragments of BPLEADER$_i$, BPAGENT$_i$ and BPSUCCESS$_i$, we provide here some comments about the messages, the state variables and the actions.

Messages. In this paragraph we describe the messages used for communication between the leader i and the agents of a round. Every message m is a tuple of elements. The messages are:

1. "Collect" messages, $m = \langle r, \text{"Collect"} \rangle_{i,j}$. This message is sent by the leader of a round to announce that a new round, with number r, has been started and at the same time to ask for information about previous rounds.

2. "Last" messages, $m = \langle r, \text{"Last"}, r', v \rangle_{j,i}$. This message is sent by an agent to respond to a "Collect" message from the leader. It provides the last round r' in which the agent has accepted a value, and the value v proposed in that round. If the agent did not accept any value in previous rounds, then v is either `nil` or the initial value of the agent and r' is $(0, j)$.

3. "Begin" messages, $m = \langle r, \text{"Begin"}, v \rangle_{i,j}$. This message is sent by the leader of round r to announce the value v of the round and at the same time to ask to accept it.

4. "Accept" messages, $m = \langle r, \text{"Accept"} \rangle_{j,i}$. This message is sent by an agent to respond to a "Begin" message from the leader. With this message an agent accepts the value proposed in the current round.

5. "OldRound" messages, $m = \langle r, \text{"OldRound"}, r' \rangle_{j,i}$. This message is sent by an agent to respond either to a "Collect" or a "Begin" message. It is sent when the agent is committed to reject round r and it informs the leader about round r', which is the higher numbered round for which the agent is committed to reject round r.

6. "Success" messages, $m = \langle \text{"Success"}, v \rangle_{i,j}$. This message is sent by the leader to broadcast the decision.

7. "Ack" messages, $m = \langle \text{"Ack"} \rangle_{j,i}$. This message is an acknowledgment, so that the leader can be sure that an agent has received the "Success" message.

We use the kind of a message to indicate any message of that kind. For example the notation $m \in \{$"Collect", "Begin"$\}$ means that m is either a "Collect" message, that is $m = \langle r,$"Collect"$\rangle$ for some r, or a "Begin" message, that is $m = \langle r,$"Begin"$,v\rangle$ for some r and v.

Automaton BPLEADER$_i$**.** Variable $Status_i$ is used to model process failures and recoveries. Variable $IamLeader_i$ keeps track of whether the process is leader. Variable $Mode_i$ is used like a program counter, to go through the steps of a round. Variable $InitValue_i$ contains the initial value of the process. Variable $Decision_i$ contains the value, if any, decided by process i. Variable $CurRnd_i$ contains the number of the round for which process i is currently the leader. Variable $HighestRnd_i$ stores the highest round number seen by process i. Variable $Value_i$ contains the value being proposed in the current round. Variable $ValFrom_i$ is the round number of the round from which $Value_i$ has been chosen (recall that a leader sets the value for its round to be equal to the value of a particular previous round, which is round $ValFrom_i$). Variable $InfoQuo_i$ contains the set of processes for which a "Last" message has been received by process i (that is, the info-quorum). Variable $AcceptQuo$ contains the set of processes for which an "Accept" message has been received by process i (that is, the accepting-quorum). We remark that in the original paper by Lamport, there is only one quorum which is fixed in the first exchange of messages between the leader and the agents, so that only processes in that quorum can accept the value being proposed. However, there is no need to restrict the set of processes that can accept the proposed value to the info-quorum of the round. Messages from processes in the info-quorum are used only to choose a consistent value for the round, and once this has been done anyone can accept that value. This improvement is also suggested in Lamport's paper [29]. Finally, variables $InMsgs_i$ and $OutMsgs_i$ are buffers used for incoming and outcoming messages.

Actions Stop$_i$ and Recover$_i$ model process failures and recoveries. Actions Leader$_i$ and NotLeader$_i$ are used to update $IamLeader_i$. Actions Send$(m)_{i,j}$ and Receive$(m)_{i,j}$ send messages to the channels and receive messages from the channels. Action Init$(v)_i$

is used by an external agent to set the initial value of process i. Action NewRound_i starts a new round. It sets the new round number by increasing the highest round number ever seen. Action Collect_i resets to the initial values all the variables that describe the status of the round being conducted and broadcasts a "Collect" message. Action $\text{GatherLast}(m)_i$ collects the information sent by agents in response to the leader's "Collect" message. This information is the number of the last round accepted by the agent and the value of that round. Upon receiving these messages, $\text{GatherLast}(m)_i$ updates, if necessary, variables Value_i and $\mathit{ValFrom}_i$. Also it updates the set of processes which eventually will be the info-quorum of the current round. Action $\text{GatherLast}(m)_i$ is executed until information is received from a majority of the processes. When "Last" messages have been collected from a majority of the processes, the info-quorum is fixed and $\text{GatherLast}(m)_i$ is no longer enabled. At this point action $\text{Gathered}(v)_i$ is enabled. If Value_i is defined then the value for the round is set, and action BeginCast_i is enabled. If Value_i is not defined (and this is possible if the leader does not have an initial value and does not receive any value in "Last" messages) the leader waits for an initial value before enabling action BeginCast_i. When an initial value is provided, action Continue_i can be executed and it sets Value_i and enables action BeginCast_i. Action BeginCast_i broadcasts a "Begin" message including the value chosen for the round. Action $\text{GatherAccept}(m)_i$ gathers the "Accept" messages. If a majority of the processes accept the value of the current round then the round is successful and GatherAccept_i sets the $\mathit{Decision}_i$ variable to the value of the current round. When variable $\mathit{Decision}_i$ has been set, action $\text{RndSuccess}(v)_i$ is enabled. Action RndSuccess_i is used to pass the decision to BPSUCCESS_i. Action $\text{GatherOldRound}(m)_i$ collects messages that inform process i that the round previously started by i is "old", in the sense that a round with a higher number has been started. Process i can update, if necessary, variable $\mathit{HighestRnd}_i$.

Automaton BPAGENT_i**.** Variable Status_i is used to model process failures and recoveries. Variable LastR_i is the round number of the latest round for which process i has sent an "Accept" message. Variable LastV_i is the value for round LastR_i. Variable

$Commit_i$ specifies the round for which process i is committed and thus specifies the set of rounds that process i must reject, which are all the rounds with round number less than $Commit_i$. We remark that when an agent commits for a round r and sends to the leader of round r a "Last" message specifying the latest round $r' < r$ in which it has accepted the proposed value, it is enough that the agent commits to not accept the value of any round r'' in between r' and r. To make the code simpler, when an agent commits for a round r, it commits to reject any round $r'' < r$. Finally, variables $InMsgs_i$ and $OutMsgs_i$ are buffers used for incoming and outcoming messages.

Actions Stop_i and Recover_i model process failures and recoveries. Actions $\text{Send}(m)_{i,j}$ and $\text{Receive}(m)_{i,j}$ send messages to the channels and receive messages from the channels. Action LastAccept_i responds to the "Collect" message sent by the leader by sending a "Last" message that gives information about the last round in which the agent has been involved. Action Accept_i responds to the "Begin" message sent by the leader. The agent accepts the value of the current round if it is not rejecting the round. In both LastAccept_i and Accept_i actions, if the agent is committed to reject the current round because of a higher numbered round, then an "OldRound" message is sent to the leader so that the leader can update the highest round number ever seen. Action $\text{Init}(v)_i$ sets to v the value of $LastV_i$ only if this variable is undefined. With this, the agent sends its initial value in a "Last" message whenever the agent has not yet accepted the value of any round.

Automaton BPSUCCESS$_i$**.** Variable $Status_i$ is used to model process failures and recoveries. Variable $IamLeader_i$ keeps track of whether the process is leader. Variable $Decision_i$ stores the decision. Variable $Acked(j)_i$ contains a boolean that specifies whether or not process j has sent an acknowledgment for a "Success" message. Variable $Prevsend_i$ records the time of the previous broadcast of the decision. Variables $LastCheck_i$, $LastSS_i$, and variables $LastSendAck(j)_i$, $LastSendSuc(j)_i$, for $j \neq i$, are used to impose the time bounds on enabled actions. Their use should be clear from the code. Variables $OutAckMsgs(j)_i$ and $OutSucMsgs(j)_i$, for $j \neq i$, are buffers for outcoming "Ack" and "Success" messages, respectively. There are no buffers for incoming messages because incoming messages are processed immediately, that is, by action Receive$(m)_{i,j}$.

Actions Stop$_i$ and Recover$_i$ model process failures and recoveries. Actions Leader$_i$ and NotLeader$_i$ are used to update $IamLeader_i$. Actions Send$(m)_{i,j}$ and Receive$(m)_{i,j}$ send messages to the channels and receive messages from the channels. Action Receive$(m)_i$ handles the receipt of "Ack" and "Success" messages. Action RndSuccess$_i$ simply takes care of updating the $Decision_i$ variable and sets a time bound for the execution of action SendSuccess$_i$. Action SendSuccess$_i$ sends the "Success" message, along with the value of $Decision_i$ to all processes for which there is no acknowledgment. It sets the time bounds for the re-sending of the "Success" message and also the time bounds $LastSendSuc(j)_i$ for the actual sending of the messages. Action Check$_i$ re-enable action SendSuccess$_i$ after an appropriate time bound. We remark that $2\ell + 2d$ is the time needed to send the "Success" message and get back an "Ack" message (see the analysis in the proof of Lemma 6.2.20).

We remark that automaton BPSUCCESS$_i$ needs to be able to measure the passage of time.

6.2.3 Partial Correctness

Let us define the system S_{BPX} to be the composition of system S_{CHA} and automaton BASICPAXOS$_i$ for each process $i \in \mathcal{I}$ (remember that BASICPAXOS$_i$ is the composition of automata BPLEADER$_i$, BPAGENT$_i$ and BPSUCCESS$_i$). In this section we prove the partial correctness of S_{BPX}: we show that in any execution of the system S_{BPX}, agreement and validity are guaranteed.

For these proofs, we augment the algorithm with a collection $\mathcal{H}$ of history variables. Each variable in $\mathcal{H}$ is an array indexed by the round number. For every round number r a history variable contains some information about round r. In particular the set $\mathcal{H}$ consists of:

`Hleader`$(r) \in \mathcal{I} \cup$ `nil`, initially `nil` (the leader of round r).

`Hvalue`$(r) \in V \cup$ `nil`, initially `nil` (the value for round r).

$\mathtt{Hfrom}(r) \in \mathcal{R} \cup \mathtt{nil}$, initially $\mathtt{nil}$ (the round from which $\mathtt{Hvalue}(r)$ is taken).

$\mathtt{Hinfquo}(r)$, subset of $\mathcal{I}$, initially $\{\}$ (the info-quorum of round r).

$\mathtt{Haccquo}(r)$, subset of $\mathcal{I}$, initially $\{\}$ (the accepting-quorum of round r).

$\mathtt{Hreject}(r)$, subset of $\mathcal{I}$, initially $\{\}$ (processes committed to reject round r).

The code fragments of automata BPLEADER$_i$ and BPAGENT$_i$ augmented with the history variables are shown in Figure 6-10. The figure shows only the actions that change history variables. Actions of BPSUCCESS$_i$ do not change history variables.

```
BPLEADER_i Actions:

input NewRound_i
  Eff: if LeaderAlive = true then
         CurRnd := HighestRnd + 1
         • Hleader(CurRnd) := i
         HighestRnd := CurRnd
         Mode := collect

output BeginCast_i
  Pre: LeaderAlive = true
       Mode = begincast
  Eff: ∀j put ⟨CurRnd,"Begin",Value⟩_{i,j}
         in OutMsgs
       • Hinfquo(CurRnd) := InfoQuo
       • Hfrom(CurRnd) := ValFrom
       • Hvalue(CurRnd) := Value
       Mode := gatheraccept

internal GatherAccept(m)_i
  Pre: LeaderAlive = true
       Mode = gatheraccept
       m =⟨r,"Accept"⟩_{j,i} ∈ InMsgs
       CurRnd = r
  Eff: remove m from InMsgs
       AcceptQuo := AcceptQuo ∪{j}
       if |AcceptQuo| > n/2 then
         Decision := Value
         • Haccquo(CurRnd):= AcceptQuo
         Mode := decide

BPAGENT_i Actions:

internal LastAccept(m)_i
  Pre: Status = alive
       m =⟨r,"Collect"⟩_{j,i} ∈ InMsgs
  Eff: remove m from InMsgs
       if r ≥ Commit then
         Commit := r
         • For all r', LastR < r' < r
         •     Hreject(r') := Hreject(r') ∪ {i}
         put ⟨r,"Last",LastR,LastV⟩_{i,j}
           in OutMsgs
       else
         put ⟨r,"OldRound",Commit⟩_{i,j}
           in OutMsgs
```

Figure 6-10: Actions of BPLEADER$_i$ and BPAGENT$_i$ for process i augmented with history variables. Only the actions that do change history variables are shown. Other actions are the same as in BPLEADER$_i$ and BPAGENT$_i$, i.e. they do not change history variables. Actions of BPSUCCESS$_i$ do not change history variables.

Initially, when no round has been started yet, all the information contained in the history variables is set to the initial values. All but $\texttt{Hreject}(r)$ history variables of round r are set by the leader of round r, thus if the round has not been started these variables remain at their initial values. More formally we have the following lemma.

Lemma 6.2.2 *In any state of an execution of S_{BPX}, if $\texttt{Hleader}(r) = \texttt{nil}$ then*

$$\begin{aligned}
\texttt{Hvalue}(r) &= \texttt{nil} \\
\texttt{Hfrom}(r) &= \texttt{nil} \\
\texttt{Hinfquo}(r) &= \{\} \\
\texttt{Haccquo}(r) &= \{\}.
\end{aligned}$$

Proof: By an easy induction. ■

Given a round r, $\texttt{Hreject}(r)$, is modified by all the processes that commit themselves to reject round r, and we know nothing about its value at the time round r is started.

Next we define some key concepts that will be instrumental in the proofs.

Definition 6.2.3 *In any state of the system S_{BPX}, a round r is said to be "dead" if $|\texttt{Hreject}(r)| \geq n/2$.*

That is, a round r is dead if at least $n/2$ of the processes are rejecting it. Hence, if a round r is dead, there cannot be a majority of the processes accepting its value, i.e., round r cannot be successful.

Definition 6.2.4 *The set $\mathcal{R}_S$ is the set $\{r \in \mathcal{R} | \texttt{Hleader}(r) \neq \texttt{nil}\}$.*

That is, $\mathcal{R}_S$ is the set of rounds that have been started. A round r is formally started as soon as its leader $\texttt{Hleader}(r)$ is defined by the NewRound$_i$ action.

Definition 6.2.5 *The set $\mathcal{R}_V$ is the set $\{r \in \mathcal{R} | \texttt{Hvalue}(r) \neq \texttt{nil}\}$.*

That is, $\mathcal{R}_V$ is the set of rounds for which the value has been chosen.

Invariant 6.2.6 *In any state s of an execution of S_{BPX}, we have that $\mathcal{R}_V \subseteq \mathcal{R}_S$.*

Indeed for any round r, if $\texttt{Hleader}(r)$ is $\texttt{nil}$, by Lemma 6.2.2 we have that $\texttt{Hvalue}(r)$ is also $\texttt{nil}$. Hence $\texttt{Hvalue}(r)$ is always set after $\texttt{Hleader}(r)$ has been set.

Next we formally define the concept of *anchored* round which is crucial to the proofs. The idea of anchored round is borrowed from [31]. Informally a round r is anchored if its value is consistent with the value chosen in any previous round r'. Consistent means that either the value of round r is equal to the value of round r' or round r' is dead. Intuitively, it is clear that if all the rounds are either anchored or dead, then agreement is satisfied.

Definition 6.2.7 *A round $r \in \mathcal{R}_V$ is said to be "anchored" if for every round $r' \in \mathcal{R}_V$ such that $r' < r$, either round r' is dead or $\texttt{Hvalue}(r') = \texttt{Hvalue}(r)$.*

Next we prove that S_{BPX} guarantees agreement, by using a sequence of invariants. The key invariant is Invariant 6.2.13 which states that all rounds are either dead or anchored. The first invariant, Invariant 6.2.8, captures the fact that when a process sends a "Last" message in response to a "Collect" message for a round r, then it commits to not vote for rounds previous to round r.

Invariant 6.2.8 *In any state s of an execution of S_{BPX}, if message $\langle r, \text{"Last"}, r'', v\rangle_{j,i}$ is in $OutMsgs_j$, then $j \in \texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$.*

Proof: We prove the invariant by induction on the length k of the execution α. The base is trivial: if $k = 0$ then $\alpha = s_0$, and in the initial state no message is in $OutMsgs_j$. Hence the invariant is vacuously true. For the inductive step assume that the invariant is true for $\alpha = s_0\pi_1 s_1 ... \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 ... \pi_k s_k \pi s$. We need to prove that the invariant is still true in s. We distinguish two cases.

CASE 1. $\langle r, \text{"Last"}, r'', v\rangle_{j,i} \in s_k.OutMsgs_j$. By the inductive hypothesis we have $j \in s_k.\texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$. Since no process is ever removed from any $\texttt{Hreject}$ set, we have $j \in s.\texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$.

CASE 2. $\langle r, \text{"Last"}, r'', v\rangle_{j,i} \notin s_k.OutMsgs_j$. Since by hypothesis we have $\langle r, \text{"Last"}, r'', v\rangle_{j,i} \in s.OutMsgs_j$, it must be that $\pi = \mathrm{LastAccept}(m)_j$, with $m = \langle r, \text{"Collect"}\rangle$ and it must be $s_k.LastR_j = r''$. Then the invariant follows by the code of LastAccept$(m)_j$ which puts process j into $\texttt{Hreject}(r')$ for all r' such that $r'' < r' < r$. ■

The next invariant states that the commitment made by an agent when sending a "Last" message is still in effect when the message is in the communication channel. This should be obvious, but to be precise in the rest of the proof we prove it formally.

Invariant 6.2.9 *In any state s of an execution of S_{BPX}, if message $\langle r, \text{"Last"}, r'', v\rangle_{j,i}$ is in $\text{CHANNEL}_{j,i}$, then $j \in \texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$.*

Proof: We prove the invariant by induction on the length k of the execution α. The base is trivial: if $k = 0$ then $\alpha = s_0$, and in the initial state no messages are in $\text{CHANNEL}_{j,i}$. Hence the invariant is vacuously true. For the inductive step assume that the invariant is true for $\alpha = s_0\pi_1 s_1 \ldots \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 \ldots \pi_k s_k \pi s$. We need to prove that the invariant is still true in s. We distinguish two cases.

Case 1. $\langle r, \text{"Last"}, r'', v\rangle_{j,i} \in s_k.\text{CHANNEL}_{j,i}$. By the inductive hypothesis we have $j \in s_k.\texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$. Since no process is ever removed from any $\texttt{Hreject}$ set, we have $j \in s.\texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$.

Case 2. $\langle r, \text{"Last"}, r'', v\rangle_{j,i} \notin s_k.\text{CHANNEL}_{j,i}$. Since by hypothesis $\langle r, \text{"Last"}, r'', v\rangle_{j,i} \in s.OutMsgs_j$, it must be that $\pi = \text{Send}(m)_{j,i}$ with $m = \langle r, \text{"Last"}, r'', v\rangle_{j,i}$. By the precondition of action $\text{Send}(m)_{j,i}$ we have that message $\langle r, \text{"Last"}, r'', v\rangle_{j,i} \in s_k.OutMsgs_j$. By Invariant 6.2.8 we have that process $j \in s_k.\texttt{Hreject}(r')$ for all r' such that $r'' < r' < r$. Since no process is ever removed from any $\texttt{Hreject}$ set, we have $j \in s.\texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$. ■

The next invariant states that the commitment made by an agent when sending a "Last" message is still in effect when the message is received by the leader. Again, this should be obvious.

Invariant 6.2.10 *In any state s of an execution of S_{BPX}, if message $\langle r, \text{"Last"}, r'', v\rangle_{j,i}$ is in $InMsgs_i$, then $j \in \texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$.*

Proof: We prove the invariant by induction on the length k of the execution α. The base is trivial: if $k = 0$ then $\alpha = s_0$, and in the initial state no messages are in

$InMsgs_i$. Hence the invariant is vacuously true. For the inductive step assume that the invariant is true for $\alpha = s_0\pi_1 s_1 ... \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 ... \pi_k s_k \pi s$. We need to prove that the invariant is still true in s. We distinguish two cases.

CASE 1. $\langle r, \text{"Last"}, r'', v\rangle_{j,i} \in s_k.InMsgs_i$. By the inductive hypothesis we have $j \in s_k.\texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$. Since no process is ever removed from any `Hreject` set, we have $j \in s.\texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$.

CASE 2. $\langle r, \text{"Last"}, r'', v\rangle_{j,i} \notin s_k.InMsgs_i$. Since by hypothesis $\langle r, \text{"Last"}, r'', v\rangle_{j,i} \in s.InMsgs_i$, it must be that $\pi = \text{Receive}(m)_{i,j}$ with $m = \langle r, \text{"Last"}, r'', v\rangle_{j,i}$. In order to execute action $\text{Receive}(m)_{i,j}$ we must have $\langle r, \text{"Last"}, r'', v\rangle_{j,i} \in s_k.\text{CHANNEL}_{j,i}$. By Invariant 6.2.9 we have $j \in s_k.\texttt{Hreject}(r')$ for all r' such that $r'' < r' < r$. Since no process is ever removed from any `Hreject` set, we have $j \in s.\texttt{Hreject}(r')$, for all r' such that $r'' < r' < r$. ■

The following invariant states that the commitment to reject smaller rounds, made by the agent is still in effect when the leader updates its information about previous rounds using the agents' "Last" messages.

Invariant 6.2.11 *In any state s of an execution S_{BPX}, if process $j \in InfoQuo_i$, for some process i, and $CurRnd_i = r$, then $\forall r'$ such that $s.ValFrom_i < r' < r$, we have that $j \in \texttt{Hreject}(r')$.*

Proof: We prove the invariant by induction on the length k of the execution α. The base is trivial: if $k = 0$ then $\alpha = s_0$, and in the initial state no process j is in $InfoQuo_i$ for any i. Hence the invariant is vacuously true. For the inductive step assume that the invariant is true for $\alpha = s_0\pi_1 s_1 ... \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 ... \pi_k s_k \pi s$. We need to prove that the invariant is still true in s. We distinguish two cases.

CASE 1. In state s_k, $j \in InfoQuo_i$, for some process i, and $CurRnd_i = r$. Then by the inductive hypothesis, in state s_k we have that $j \in \texttt{Hreject}(r')$, for all r' such that $s_k.ValFrom_i < r' < r$. Since no process is ever removed from any `Hreject` set and, as long as $CurRnd_i$ is not changed, variable $ValFrom_i$ is never decreased, then also in state s we have that $j \in \texttt{Hreject}(r')$, for all r' such that $s.ValFrom_i < r' < r$.

CASE 2. In state s_k, it is not true that $j \in \mathit{InfoQuo}_i$, for some process i, and $\mathit{CurRnd}_i = r$. Since in state s it holds that $j \in \mathit{InfoQuo}_i$, for some process i, and $\mathit{CurRnd}_i = r$, it must be the case that $\pi = \text{GatherLast}(m)_i$ with $m = \langle r, \text{“Last”}, r'', v \rangle_{j,i}$. Notice that, by the precondition of GatherLast$(m)_i$, $m \in \mathit{InMsgs}_i$. Hence, by Invariant 6.2.10 we have that $j \in \mathtt{Hreject}(r')$, for all r' such that $r'' < r' < r$. By the code of the GatherLast$(m)_i$ action we have that $\mathit{ValFrom}_i \geq r''$. Whence the invariant is proved. ■

The following invariant is basically the previous one stated when the leader has fixed the info-quorum.

Invariant 6.2.12 *In any state of an execution of S_{BPX}, if $j \in \mathtt{Hinfquo}(r)$ then $\forall r'$ such that $\mathtt{Hfrom}(r) < r' < r$, we have that $j \in \mathtt{Hreject}(r')$.*

Proof: We prove the invariant by induction on the length k of the execution α. The base is trivial: if $k = 0$ then $\alpha = s_0$, and in the initial state we have that for every round r, $\mathtt{Hleader}(r) = \mathtt{nil}$ and thus by Lemma 6.2.2 there is no process j in $\mathtt{Hinfquo}(r)$. Hence the invariant is vacuously true. For the inductive step assume that the invariant is true for $\alpha = s_0\pi_1 s_1 ... \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 ... \pi_k s_k \pi s$. We need to prove that the invariant is still true in s. We distinguish two cases.

CASE 1. In state s_k, $j \in \mathtt{Hinfquo}(r)$. By the inductive hypothesis, in state s_k we have that $j \in \mathtt{Hreject}(r')$, for all r' such that $\mathtt{Hfrom}(r) < r' < r$. Since no process is ever removed from any $\mathtt{Hreject}$ set, then also in state s we have that $j \in \mathtt{Hreject}(r')$, for all r' such that $\mathtt{Hfrom}(r) < r' < r$.

CASE 2. In state s_k, $j \notin \mathtt{Hinfquo}(r)$. Since in state s, $j \in \mathtt{Hinfquo}(r)$, it must be the case that action π puts j in $\mathtt{Hinfquo}(r)$. Thus it must be $\pi = \text{BeginCast}_i$ for some process i, and it must be $s_k.\mathit{CurRnd}_i = r$ and $j \in s_k.\mathit{InfoQuo}_i$. Since action BeginCast$_i$ does not change CurRnd_i and $\mathit{InfoQuo}_i$ we have that $s.\mathit{CurRnd}_i = r$ and $j \in s.\mathit{InfoQuo}_i$. By Invariant 6.2.11 we have that $j \in \mathtt{Hreject}(r')$ for all r' such that $s.\mathit{ValFrom}_i < r' < r$. By the code of BeginCast$_i$ we have that $\mathtt{Hfrom}(r) = s.\mathit{ValFrom}_i$. ■

We are now ready to prove the main invariant.

Invariant 6.2.13 *In any state of an execution of S_{BPX}, any non-dead round $r \in \mathcal{R}_V$ is anchored.*

Proof: We proceed by induction on the length k of the execution α. The base is trivial. When $k = 0$ we have that $\alpha = s_0$ and in the initial state no round has been started yet. Thus $\texttt{Hleader}(r) = \texttt{nil}$ and by Lemma 6.2.2 we have that $\mathcal{R}_V = \{\}$ and thus the assertion is vacuously true. For the inductive step assume that the assertion is true for $\alpha = s_0\pi_1 s_1 \ldots \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 \ldots \pi_k s_k \pi s$. We need to prove that, for every possible action π the assertion is still true in state s. First we observe that the definition of "dead" round depends only upon the history variables and that the definition of "anchored" round depends upon the history variables and the definition of "dead" round. Thus the definition of "anchored" depends only on the history variables. Hence actions that do not modify the history variables cannot affect the truth of the assertion. The actions that change history variables are:

1. $\pi = \mathrm{NewRound}_i$
2. $\pi = \mathrm{BeginCast}_i$
3. $\pi = \mathrm{GatherAccept}(m)_i$
4. $\pi = \mathrm{LastAccept}(m)_i$

CASE 1. Assume $\pi = \mathrm{NewRound}_i$. This action sets the history variable $\texttt{Hleader}(r)$, where r is the round number of the round being started by process i. The new round r does not belong to $\mathcal{R}_V$ since $\texttt{Hvalue}(r)$ is still undefined. Thus the assertion of the lemma cannot be contradicted by this action.

CASE 2. Assume $\pi = \mathrm{BeginCast}_i$. Action π sets $\texttt{Hvalue}(r)$, $\texttt{Hfrom}(r)$ and $\texttt{Hinfquo}(r)$ where $r = s_k.\mathit{CurRnd}$. Round r belongs to $\mathcal{R}_V$ in the new state s. In order to prove that the assertion is still true it suffices to prove that round r is anchored in state s and any round r', $r' > r$ is still anchored in state s. Indeed rounds with round number less than r are still anchored in state s, since the definition of anchored for a given round involves only rounds with smaller round numbers.

First we prove that round r is anchored. From the precondition of $\mathrm{BeginCast}_i$ we have that $\mathtt{Hinfquo}(r)$ contains more than $n/2$ processes; indeed variable *Mode* is equal to `begincast` only if the cardinality of *InfoQuo* is greater than $n/2$. Using Invariant 6.2.12 for each process j in $s.\mathtt{Hinfquo}(r)$, we have that for every round r', such that $s.\mathtt{Hfrom}(r) < r' < r$, there are more than $n/2$ processes in the set $\mathtt{Hreject}(r')$, which means that every round r', $s.\mathtt{Hfrom}(r) < r' < r$, is dead. Moreover, by the code of π we have that $s.\mathtt{Hfrom}(r) = s_k.\mathit{ValFrom}$ and $s.\mathtt{Hvalue}(r) = s_k.\mathit{Value}$. From the code (see action $\mathrm{GatherLast}_i$) it is immediate that *Value* is the value of round *ValFrom*. In particular we have that $s_k.\mathit{Value} = s_k.\mathtt{Hvalue}(s_k.Value)$. Hence we have $s.\mathtt{Hvalue}(r) = s.\mathtt{Hvalue}(s.\mathtt{Hfrom}(r))$. Finally we notice that round $\mathtt{Hfrom}(r)$ is anchored (any round previous to r is still anchored in state s) and thus we have that any round $r' < r$ is either dead or such that $s.\mathtt{Hvalue}(s.\mathtt{Hfrom}(r)) = s.\mathtt{Hvalue}(r')$. Hence for any round $r' < r$ we have that either round r' is dead or that $s.\mathtt{Hvalue}(r) = s.\mathtt{Hvalue}(r')$. Thus round r is anchored in state s.

Finally, we need to prove that any non-dead round r', $r' > r$ that was anchored in s_k is still anchored in s. Since action $\mathrm{BeginCast}_i$ modifies only history variables for round r, we only need to prove that in state s, $\mathtt{Hvalue}(r') = \mathtt{Hvalue}(r)$. Let r'' be equal to $\mathtt{Hfrom}(r)$. Since r' is anchored in state s_k we have that $s_k.\mathtt{Hvalue}(r') = s_k.\mathtt{Hvalue}(r'')$. Again because $\mathrm{BeginCast}_i$ modifies only history variables for round r, we have that $s.\mathtt{Hvalue}(r') = s.\mathtt{Hvalue}(r'')$. But we have proved that round r is anchored in state s and thus $s.\mathtt{Hvalue}(r) = s.\mathtt{Hvalue}(r'')$. Hence $s.\mathtt{Hvalue}(r') = s.\mathtt{Hvalue}(r)$.

Case 3. Assume $\pi = \mathrm{GatherAccept}(m)_i$. This action modifies only variable `Haccquo`, which is not involved in the definition of anchored. Thus this action cannot make the assertion false.

Case 4. Assume $\pi = \mathrm{LastAccept}(m)_i$. This action modifies `Hinfquo` and `Hreject`. Variable `Hinfquo` is not involved in the definition of anchored. Action $\mathrm{LastAccept}(m)_i$ may put process i in `Hreject` of some rounds and this, in turn, may make those rounds dead. However this cannot make false the assertion; indeed if a round r was anchored in s_k it is still anchored when another round becomes dead. ■

The next invariant follows from the previous one and gives a more direct statement about the agreement property.

Invariant 6.2.14 *In any state of an execution of S_{BPX}, all the Decision variables that are not* `nil`*, are set to the same value.*

Proof: We prove the invariant by induction on the length k of the execution α. The base of the induction is trivially true: for $k = 0$ we have that $\alpha = s_0$ and in the initial state all the $Decision_i$ variables are undefined.
Assume that the assertion is true for $\alpha = s_0\pi_1 s_1 \ldots \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 \ldots \pi_k s_k \pi s$. We need to prove that, for every possible action π the assertion is still true in state s. Clearly the only actions which can make the assertion false are those that set $Decision_i$, for some process i. Thus we only need to consider actions GatherAccept($\langle r$, "Accept"$\rangle)_i$ and actions RndSuccess$(v)_i$ and Receive($\langle$"Success", $v\rangle)_{i,j}$ of automaton BPSUCCESS$_i$.

CASE 1. Assume π = GatherAccept($\langle r$, "Accept"$\rangle)_i$. This action sets $Decision_i$ to $\mathtt{Hvalue}(r)$. If all $Decision_j$, $j \neq i$, are undefined then $Decision_i$ is the first decision and the assertion is still true. Assume there is only one $Decision_j$ already defined. Let $Decision_j = \mathtt{Hvalue}(r')$ for some round r'. By Invariant 6.2.13, rounds r and r' are anchored and thus we have that $\mathtt{Hvalue}(r') = \mathtt{Hvalue}(r)$. Whence $Decision_i = Decision_j$. If there are some $Decision_j$, $j \neq i$, which are already defined, then by the inductive hypothesis they are all equal. Thus, the lemma follows.

CASE 2. Assume π = RndSuccess$(v)_i$. This action sets $Decision_i$ to v. By the code, value v is equal to the $Decision_j$ of some other process. The lemma follows by the inductive hypothesis.

CASE 2. Assume π = Receive($\langle$"Success",$v\rangle)_i$. This action sets $Decision_i$ to v. It is easy to see (by the code) that the value sent in a "Success" message is always the *Decision* of some process. Thus we have that $Decision_i$ is equal to $Decision_j$ for some other process j and the lemma follows by the inductive hypothesis. ■

Finally we can prove that agreement is satisfied.

Theorem 6.2.15 *In any execution of the system S_{BPX}, agreement is satisfied.*

Proof: Immediate from Invariant 6.2.14. ■

Validity is easier to prove since the value proposed in any round comes either from a value supplied by an $\text{Init}(v)_i$ action or from a previous round.

Invariant 6.2.16 *In any state of an execution α of S_{BPX}, for any $r \in \mathcal{R}_V$ we have that* `Hvalue`$(r) \in V_\alpha$.

Proof: We proceed by induction on the length k of the execution α. The base of the induction is trivially true: for $k = 0$ we have that $\alpha = s_0$ and in the initial state all the `Hvalue` variables are undefined.
Assume that the assertion is true for $\alpha = s_0\pi_1 s_1 ... \pi_k s_k$ and consider the execution $s_0\pi_1 s_1 ... \pi_k s_k \pi s$. We need to prove that, for every possible action π the assertion is still true in state s. Clearly the only actions that can make the assertion false are those that modify `Hvalue`. The only action that modifies `Hvalue` is BeginCast. Thus, assume $\pi = \text{BeginCast}_i$. This action sets `Hvalue`(r) to Value_i. We need to prove that all the values assigned to Value_i are in the set V_α. Variable Value_i is modified by actions NewRound_i and $\text{GatherLast}(m)_i$. We can easily take care of action NewRound_i because it simply sets Value_i to be $\mathit{InitValue}_i$ which is obviously in V_α. Thus we only need to worry about $\text{GatherLast}(m)_i$ actions. A $\text{GatherLast}(m)_i$ action sets variable Value_i to the value specified into the "Last" message if that value is not `nil`. The value specified into any "Last" message is either `nil` or the value `Hvalue`(r') of a previous round r'; by the inductive hypothesis we have that `Hvalue`(r') belongs to V_α. ■

Invariant 6.2.17 *In any state of an execution of S_{BPX}, all the Decision variables that are not undefined are set to some value in V_α.*

Proof: A variable *Decision* is always set to be equal to `Hvalue`(r) for some r. Thus the invariant follows from Invariant 6.2.16. ■

Theorem 6.2.18 *In any execution of the system S_{BPX}, validity is satisfied.*

Proof: Immediate from Invariant 6.2.17. ■

6.2.4 Analysis of S_{BPX}

In this section we analyze the performance of S_{BPX}. Since termination is not guaranteed by S_{BPX} in this section we provide a performance analysis (Lemma 6.2.24) assuming that a successful round is conducted. Then in Section 6.4, Theorem 6.4.2 provides the performance analysis of S_{PAX}, which, in a nice execution fragment, guarantees termination.

Let us begin by making precise the meaning of the expressions "the start (end) of a round".

Definition 6.2.19 *In an execution fragment whose states are all unique-leader states with process i being the unique leader, the "start" of a round is the execution of action NewRound$_i$ and the "end" of a round is the execution of action RndSuccess$_i$.*

A round is *successful* if it ends, that is, if the RndSuccess$_i$ action is executed by the leader i. Moreover we say that a process i *reaches* its decision when automaton BPSUCCESS$_i$ sets its *Decision*$_i$ variable. We remark that, in the case of a leader, the decision is actually reached when the leader knows that a majority of the processes have accepted the value being proposed. This happens in action GatherAccept$(m)_i$ of BPLEADER$_i$. However, to be precise in our proofs, we consider the decision reached when the variable *Decision*$_i$ of BPSUCCESS$_i$ is set; for the leader this happens exactly at the end of a successful round. Notice that the Decide$(v)_i$ action, which communicates the decision v of process i to the external environment, is executed within ℓ time from the point in time when process i reaches the decision, provided that the execution is regular (in a regular execution actions are executed within the expected time bounds).

The following lemma states that once a round has ended, if the execution is stable, the decision is reached by all the alive processes within linear (in the number of processes) time.

Lemma 6.2.20 *If an execution fragment α of the system S_{BPX}, starting in a reachable state s and lasting for more than $3\ell + 2d$ time, is stable and unique-leader, with process i leader, and process i reaches a decision in state s, then by time $3\ell+2d$, every* *alive process $j \neq i$ has reached a decision, and the leader i has $Acked(j)_i = \texttt{true}$ for every alive process $j \neq i$.*

Proof: First notice that S_{BPX} is the composition of $\text{CHANNEL}_{i,j}$ and other automata. Hence, by Theorem 2.6.10 we can apply Lemma 3.2.3. Let $\mathcal{J}$ be the alive processes $j \neq i$ such that $Acked(j)_i = \texttt{false}$. If $\mathcal{J}$ is empty then the lemma is trivially true. Hence assume $\mathcal{J} \neq \{\}$.

By assumption, the action that brings the system into state s is action RndSuccess_i (the leader reaches a decision in state s). Hence action SendSuccess_i is enabled. By the code of BPSUCCESS_i, action SendSuccess_i is executed within ℓ time. This action puts a "Success" message for each process $j \in \mathcal{J}$ into $OutSucMsgs(j)_i$. By the code of BPSUCCESS_i, each of these messages is put on $\text{CHANNEL}_{i,j}$, i.e., action $\text{Send}(\langle\text{"Success"},v\rangle)_{i,j}$ is executed, within ℓ time. By Lemma 3.2.3 each alive process $j \in \mathcal{J}$ receives the "Success" message, i.e., executes a $\text{Receive}(\langle\text{"Success"},v\rangle)_{i,j}$ action, within d time. This action sets $Decision_j$ to v and puts an "Ack" message into $OutAckMsgs(i)_j$. By the code of BPSUCCESS_j, this "Ack" message is put on $\text{CHANNEL}_{j,i}$, i.e., action $\text{Send}(\text{"Ack"})_{j,i}$ is executed, within ℓ time, for every process j. By Lemma 3.2.3 the leader i receives the "Ack" message, i.e., executes a $\text{Receive}(\langle\text{"Ack"}\rangle)_{j,i}$ action, within d time, for each process j. This action sets $Acked(j)_i = \texttt{true}$.

Summing up the time bounds we get the lemma. ∎

In the following we are interested in the time analysis from the start to the end of a successful round. We consider a unique-leader execution fragment α, with process i leader, and such that the leader i has started a round by the first state of α (that is, in the first state of α, $CurRnd_i = r$ for some round number r).

We remark that in order for the leader to execute step 3 of a round, i.e., action BeginCast_i, it is necessary that $Value_i$ be defined. If the leader does not have an initial value and no agent sends a value in a "Last" message, variable $Value_i$ is not defined. In this case the leader needs to wait for the execution of the $\text{Init}(v)_i$ to set a value to propose in the round (see action Continue_i). Clearly the time analysis depends on the time of occurrence of the $\mathrm{Init}(v)_i$. To deal with this we use the following definition.

Definition 6.2.21 *Given an execution fragment α, we define t^i_α to be 0, if variable $InitValue_i$ is defined in the first state of α; the time of occurrence of action $Init(v)_i$, if variable $InitValue_i$ is undefined in the first state of α and action $Init(v)_i$ is executed in α; ∞, if variable $InitValue_i$ is undefined in the first state of α and no $Init(v)_i$ action is executed in α. Moreover, we define T^i_α to be $\max\{7\ell + 2d, t^i_\alpha + 2\ell\}$.*

Informally, the above definition of T^i_α gives the time, counted from the beginning of a round, by which a $\mathrm{BeginCast}_i$ action is expected to be executed, assuming that the execution α is stable and the round being conducted is successful. More formally we have the following lemma.

Lemma 6.2.22 *Suppose that for an execution fragment α of the system S_{BPX}, starting in a reachable state s in which $s.Decision = \texttt{nil}$, it holds that:*

(i) α is stable;

(ii) α is a unique-leader execution, with process i leader;

(iii) α lasts for more than T^i_α;

(iv) the action that brings the system into state s is action $NewRound_i$ for some round r;

(v) round r is successful.

Then we have that action $BeginCast_i$ for round r is executed within time T^i_α of the beginning of α.

Proof: First notice that S_{BPX} is the composition of $\mathrm{CHANNEL}_{i,j}$ and other automata. Hence, by Theorem 2.6.10 we can apply Lemmas 2.5.4 and 3.2.3. Since the execution is stable, it is also regular, and thus by Lemma 2.5.4 actions of $\mathrm{BPLEADER}_i$ and $\mathrm{BPAGENT}_i$ are executed within ℓ time and by Lemma 3.2.3 messages are delivered within d time.

Action NewRound$_i$ enables action Collect$_i$ which is executed in at most ℓ time. This action puts "Collect" messages, one for each agent j, into *OutMsgs*$_i$. By the code of BPLEADER$_i$ (see tasks and bounds) each one of these messages is sent on CHANNEL$_{i,j}$ i.e., action Send$_{i,j}$ is executed for each of these messages, within ℓ time. By Lemma 3.2.3 a "Collect" message is delivered to each agent j, i.e., action Receive$_{i,j}$ is executed, within d time. Then it takes ℓ time for an agent to execute action LastAccept$_j$ which puts a "Last" message in *OutMsgs*$_j$. By the code of BPAGENT$_i$ (see tasks and bounds) it takes additional ℓ time to execute action Send$_{j,i}$ to send the "Last" message on CHANNEL$_{j,i}$. By Lemma 3.2.3, this "Last" message is delivered to the leader, i.e., action Receive$_{j,i}$ is executed, within additional d time. By the code of BPLEADER$_i$ (see tasks and bounds) each one of these messages is processed by GatherLast$_i$ within ℓ time. Action Gathered$_i$ is executed within additional ℓ time.

At this point there are two possible cases: (i) *Value*$_i$ is defined and (ii) *Value*$_i$ is not defined. In case (i), action BeginCast$_i$ is enabled and is executed within ℓ time. Summing up the times considered so far we have that action BeginCast$_i$ is executed within $7\ell + 2d$ time from the start of the round. In case (ii), action Continue$_i$ is executed within ℓ time of the execution of action Continue$_i$, and thus by time $t^i_\alpha + \ell$. This action enables action BeginCast$_i$ which is executed within additional ℓ time. Hence action BeginCast$_i$ is executed by time $t^i_\alpha + 2\ell$. Putting together the two cases we have that action BeginCast$_i$ is executed by time $\max\{7\ell + 2d, t^i_\alpha + 2\ell\}$.

Hence we have proved that action BeginCast$_i$ is executed in α by time T^i_α. ■

Next lemma gives a bound for the time that elapses between the execution of the BeginCast$_i$ action and the RndSuccess$_i$ action for a successful round in a stable execution fragment.

Lemma 6.2.23 *Suppose that for an execution fragment α of the system S_{BPX}, starting in a reachable state s in which $s.Decision =$* `nil`*, it holds that:*

(i) *α is stable;*

(ii) *α is a unique-leader execution, with process i leader;*

(iii) *α lasts for more than $5\ell + 2d$ time;*

(iv) *the action that brings the system into state s is action BeginCast$_i$ for some round r;*

(v) *round r is successful.*

Then we have that action RndSuccess$_i$ is performed by time $5\ell+2d$ from the beginning of α.

Proof: First notice that S_{BPX} is the composition of CHANNEL$_{i,j}$ and other automata. Hence, by Theorem 2.6.10 we can apply Lemmas 2.5.4 and 3.2.3. Since the execution is stable, it is also regular, and thus by Lemma 2.5.4 actions of BPLEADER$_i$ and BPAGENT$_i$ are executed within ℓ time and by Lemma 3.2.3 messages are delivered within d time.

Action BeginCast$_i$ puts "Begin" messages for round r in *OutMsgs$_i$*. By the code of BPLEADER$_i$ (see tasks and bounds) each one of these messages is put on CHANNEL$_{i,j}$ by means of action Send$_{i,j}$ in at most ℓ time. By Lemma 3.2.3 a "Begin" message is delivered to each agent j, i.e., action Receive$_{i,j}$ is executed, within d time. By the code of BPAGENT$_j$ (see tasks and bounds) action Accept$_j$ is executed within ℓ time. This action puts an "Accept" message in *OutMsgs$_j$*. By the code of BPAGENT$_j$ the "Accept" message is put on CHANNEL$_{j,i}$, i.e., action Send$_{j,i}$ for this message is executed, within ℓ time. By Lemma 3.2.3 the message is delivered, i.e., action Receive$_{j,i}$ for that message is executed, within d time. By the code of BPLEADER$_i$ action GatherAccept$_i$ is executed for a majority of the "Accept" messages within additional ℓ time. At this point variable *Decision$_i$* is defined and action RndSuccess$_i$ is executed within ℓ time. Summing up all the times we have that the round ends within $5\ell + 2d$. ■

We can now easily prove a time bound on the time needed to complete a round.

Lemma 6.2.24 *Suppose that for an execution fragment α of the system S_{BPX}, starting in a reachable state s in which $s.Decision =$ `nil`, it holds that:*

(i) *α is stable;*

(ii) *α is a unique-leader execution, with process i leader;*

(iii) *α lasts for more than $T^i_\alpha + 5\ell + 2d$;*

(iv) *the action that brings the system into state s is action $NewRound_i$ for some round r;*

(v) *round r is successful.*

Then we have that action $BeginCast_i$ for round r is executed within time T^i_α of the beginning of α and action $RndSuccess_i$ is executed by time $T^i_\alpha + 5\ell + 2d$ of the beginning of α.

Proof: Follows from Lemmas 6.2.22 and 6.2.23 ■

The previous lemma states that in a stable execution a successful round is conducted within some time bound. However it is possible that even if the system executes nicely from some point in time on, no successful round is conducted and to have a successful round a new round must be started. We take care of this problem in the next section. We will use a more refined version of Lemma 6.2.24; this refined version replaces condition (v) with a weaker requirement. This weaker requirement is enough to prove that the round is successful.

Lemma 6.2.25 *Suppose that for an execution fragment α of S_{BPX}, starting in a reachable state s in which $s.Decision =$ `nil`, it holds that:*

(i) *α is nice;*

(ii) *α is a unique-leader execution, with process i leader;*

(iii) *α lasts for more than $T^i_\alpha + 5\ell + 2d$ time;*

(iv) *the action that brings the system into state s is action $NewRound_i$ for some round r;*

(v) *there exists a set $\mathcal{J} \subseteq \mathcal{I}$ of processes such that every process in $\mathcal{J}$ is alive and $\mathcal{J}$ is a majority, for every $j \in \mathcal{J}$, $s.Commit_j \leq r$ and in state s for every $j \in \mathcal{J}$ and $k \in \mathcal{I}$, $\textsc{channel}_{k,j}$ and $InMsgs_j$ do not contain any "Collect" message belonging to any round $r' > r$.*

Then we have that action $BeginCast_i$ is performed by time T^i_α and action $RndSuccess_i$ is performed by time $T^i_\alpha + 5\ell + 2d$ from the beginning of α.

Proof: Process i sends a "Collect" message which is delivered to all the alive voters. All the alive voters, and thus all the processes in $\mathcal{J}$, respond with "Last" messages which are delivered to the leader. No process $j \in \mathcal{J}$ can be committed to reject round r. Indeed, by assumption, process j is not committed to reject round r in state s and process j cannot commit to reject round r. The latter is due to the fact that in state s no message that can cause process j to commit to reject round r is either in $InMsgs_j$ nor in any channel to process j, and in α the only leader is i, which only sends messages belonging to round r. Since $\mathcal{J}$ is a majority, the leader receives at least a majority of "Last" messages and thus it is able to proceed with the next step of the round. The leader sends a "Begin" message which is delivered to all the alive voters. All the alive voters, and thus all the processes in $\mathcal{J}$, respond with "Accept" messages since they are not committed to reject round r. Since $\mathcal{J}$ is a majority, the leader receives at least a majority of "Accept" messages. Therefore given that α lasts for enough time round r is successful.

Since round r is successful, the lemma follows easily from Lemma 6.2.24. ■

6.3 Automaton S_{PAX}

To reach consensus using S_{BPX}, rounds must be started by an external agent by means of the $\mathrm{NewRound}_i$ action that makes process i start a new round. In this section we provide automata $\textsc{starteralg}_i$ that start new round. Composing $\textsc{starteralg}_i$ with S_{BPX} we obtain S_{PAX}.

The system S_{BPX} guarantees that running rounds does not violate agreement and validity, even if rounds are started by many processes. However since running a new round may prevent a previous one from succeeding, initiating too many rounds is not a good idea. The strategy used to initiate rounds is to have a leader election algorithm and let the leader initiate new rounds until a round is successful. We exploit the robustness of BASICPAXOS in order to use the sloppy leader elector provided in Section 5. As long as the leader elector does not provide exactly one leader, it is possible that no round is successful, however agreement and validity are always guaranteed. Moreover, when the leader elector provides exactly one leader, if the system S_{BPX} is executing a nice execution fragment then a round is successful.

Automaton STARTERALG$_i$ takes care of the problem of starting new rounds. This automaton interacts with LEADERELECTOR$_i$ by means of the Leader$_i$ and NotLeader$_i$ actions and with BASICPAXOS$_i$ by means of the NewRound$_i$, Gathered$(v)_i$, Continue$_i$ and RndSuccess$(v)_i$ actions. Figure 6-1, given at the beginning of the section, shows the interaction of the STARTERALG$_i$ automaton with the other automata.

Automaton STARTERALG$_i$ does the following. Whenever process i becomes leader, the STARTERALG$_i$ automaton starts a new round by means of action NewRound$_i$. Moreover the automaton checks that action BeginCast$_i$ is executed within the expected time bound (given by Lemma 6.2.24). If BeginCast$_i$ is not executed within the expected time bound, then STARTERALG$_i$ starts a new round. Similarly once BeginCast$_i$ has been executed, the automaton checks that action RndSuccess$(v)_i$ is executed within the expected time bound (given by Lemma 6.2.24). Again, if such an action is not executed within the expected time bound, STARTERALG$_i$ starts a new round. We remark that to check for the execution of BeginCast$_i$, the automaton actually checks for the execution of action Gathered$(v)_i$. This is because the expected time of execution of BeginCast$_i$ depends on whether an initial value is already available when action Gathered$(v)_i$ is executed. If such a value is available when Gathered$(v)_i$ is executed then BeginCast$_i$ is enabled and is expected to be executed within ℓ time of the execution of Gathered$(v)_i$. Otherwise the leader has to wait for the execution of action Init$(v)_i$ which enables action Continue$_i$ and action BeginCast$_i$

STARTERALG$_i$

Signature:

Input:	Leader$_i$, NotLeader$_i$, Stop$_i$, Recover$_i$, Gathered$(v)_i$, Continue$_i$, RndSuccess$(v)_i$
Internal:	CheckGathered$_i$, CheckRndSuccess$_i$
Output:	NewRound$_i$
Time-passage:	$\nu(t)$

State:

$Clock \in \mathbb{R}$	initially arbitrary	$DlineSuc \in \mathbb{R} \cup \{\infty\}$	initially `nil`
$Status \in \{$`alive`,`stopped`$\}$	initially `alive`	$DlineGat \in \mathbb{R} \cup \{\infty\}$	initially `nil`
IamLeader, a boolean	initially `false`	$LastNR \in \mathbb{R} \cup \{\infty\}$	initially ∞
Start, a boolean	initially `false`	*RndSuccess*, a boolean	initially `false`

Actions:

input Stop$_i$
Eff: $Status :=$ `stopped`

input Recover$_i$
Eff: $Status :=$ `alive`

input Leader$_i$
Eff: if $Status =$ `alive` then
  if $IamLeader =$ `false` then
   $IamLeader :=$ `true`
   if $RndSuccess =$ `false` then
    $Start :=$ `true`
    $DlineGat := \infty$
    $DlineSuc := \infty$
    $LastNR := Clock + \ell$

input NotLeader$_i$
Eff: if $Status =$ `alive` then
  $LastNR := \infty$
  $DlineSus := \infty$
  $DlineGat := \infty$
  $IamLeader :=$ `false`

output NewRound$_i$
Pre: $Status =$ `alive`
  $IamLeader =$ `true`
  $Start =$ `true`
Eff: $Start :=$ `false`
  $DlineGat := Clock + 6\ell + 2d$
  $LastNR := \infty$

input Gathered$(v)_i$
Eff: if $Status =$ `alive` then
  $DlineGat := \infty$
  if $v \neq$ `nil` then
   $DlineSuc := Clock + 6\ell + 2d$

input Continue$_i$
Eff: if $Status =$ `alive` then
   $DlineSuc := Clock + 6\ell + 2d$

input RndSuccess$(v)_i$
Eff: if $Status =$ `alive` then
   $RndSuccess :=$ `true`
   $DlineGat := \infty$
   $DlineSuc := \infty$
   $LastNR := \infty$

internal CheckGathered$_i$
Pre: $Status =$ `alive`
  $IamLeader =$ `true`
  $DlineGat \neq$ `nil`
  $Clock > DlineGat$
Eff: $DlineGat := \infty$
  $Start :=$ `true`
  $LastNR := Clock + \ell$

internal CheckRndSuccess$_i$
Pre: $Status =$ `alive`
  $IamLeader =$ `true`
  $DlineSuc \neq$ `nil`
  $Clock > DlineSuc$
Eff: $DlineSuc := \infty$
  $Start :=$ `true`
  $LastNR := Clock + \ell$

time-passage $\nu(t)$
Pre: $Status =$ `alive`
Eff: Let t' be s.t. $Clock + t' \leq LastNR$
   and $Clock + t' \leq DlineGat + \ell$
   and $Clock + t' \leq DlineSuc + \ell$
  $Clock := Clock + t'$

Figure 6-11: Automaton STARTERALG for process i

is expected to be executed within ℓ time of the execution of Continue_i.

In addition to the code we provide some comments about the state variables and the actions. Variables $\mathit{IamLeader}_i$ and Status_i are self-explanatory. Variable Start_i is true when a new round needs to be started. Variable $\mathit{RndSuccess}_i$ is true when a decision has been reached. Variables $\mathit{DlineGat}_i$ and $\mathit{DlineSuc}_i$ are used to check for the execution of actions $\text{Gathered}(v)_i$ and $\text{RndSuccess}(v)_i$. They are also used, together with variable LastNR_i, to impose time bounds on enabled actions.

Automaton STARTERALG_i updates variable $\mathit{IamLeader}_i$ according to the input actions Leader_i and NotLeader_i and executes internal and output actions whenever it is the leader. Variable *Start* is used to start a new round and it is set either when a Leader_i action changes the leader status *IamLeader* from `false` to `true`, that is, when the process becomes leader or when the expected time bounds for the execution of actions $\text{Gathered}(v)_i$ and $\text{RndSuccess}(v)_i$ elapse without the execution of these actions. Variable $\mathit{RndSuccess}_i$ is updated by the input action $\text{RndSuccess}(v)_i$. Action NewRound_i starts a new round. Actions CheckGathered_i and CheckRndSuccess_i check, respectively, whether actions $\text{Gathered}(v)_i$ and $\text{RndSuccess}(v)_i$ are executed within the expected time bounds. Using an analysis similar to the one done in the proof of Lemma 6.2.22 we have that action $\text{Gathered}(v)_i$ is supposed to be executed within $6\ell + 2d$ time of the start of the round. The time bound for the execution of action $\text{RndSuccess}(v)_i$ depends on whether the leader has to wait for an $\text{Init}(v)_i$ event. However by Lemma 6.2.23 action $\text{RndSuccess}(v)_i$ is expected to be executed within $5\ell + 2d$ time from the time of occurrence of action BeginCast_i and action BeginCast_i is executed either within ℓ time of the execution of action $\text{Gathered}(v)_i$, if an initial value is available when this action is executed, or else within ℓ time of the execution of action Continue_i. Hence actions $\text{Gathered}(v)_i$ and Continue_i both set a deadline of $6\ell + 2d$ for the execution of action $\text{RndSuccess}(v)_i$. Actions CheckGathered_i and CheckRndSuccess_i start a new round if the above deadlines expire.

6.4 Correctness and analysis of S_{PAX}

Even in a nice execution fragment a round may not reach success. This is possible when agents are committed to reject the first round started in the nice execution fragment because they are committed for higher numbered rounds started before the beginning of the nice execution fragment. However in such a case a new round is started and there is nothing that can prevent the success of the new round. Indeed in the newly started round, alive processes are not committed for higher numbered rounds since during the first round they inform the leader of the round number for which they are committed and the leader, when starting a new round, always uses a round number greater than any round number ever seen. In this section we will prove that in a long enough nice execution fragment termination is guaranteed.

Remember that S_{PAX} is the system obtained by composing system S_{LEA} with one automaton BASICPAXOS$_i$ and one automaton STARTERALG$_i$ for each process $i \in \mathcal{I}$. Since this system contains as a subsystem the system S_{BPX}, it guarantees agreement and validity. However, in a long enough nice execution fragment of S_{PAX} termination is achieved, too.

The following lemma states that in a long enough nice, unique-leader execution, the leader reaches a decision. We recall that $T^i_\alpha = \max\{7\ell + 2d, t^i_\alpha + 2\ell\}$ and that t^i_α is the time of occurrence of action $\text{Init}(v)_i$ in α (see Definition 6.2.21).

Lemma 6.4.1 *Suppose that for an execution fragment α of S_{PAX}, starting in a reachable state s in which $s.Decision = \texttt{nil}$, it holds that*

(i) α is nice;

(ii) α is a unique-leader execution, with process i leader;

(iii) α lasts for more than $T^i_\alpha + 20\ell + 7d$ time.

Then by time $T^i_\alpha + 20\ell + 7d$ the leader i has reached a decision.

Proof: First we notice that system S_{PAX} contains as subsystem S_{BPX}; hence by using Theorem 2.6.10, the projection of α on the subsystem S_{BPX} is actually an execution of S_{BPX} and thus Lemmas 6.2.24 and 6.2.25 are still true in α.

For simplicity, in the following we assume that $T^i_\alpha = 0$, i.e., that process i has executed an $\mathrm{Init}(v)_i$ action before α. At the end of each case we consider, we will add T^i_α to the time bound to take into account the possibility that process i has to wait for an $\mathrm{Init}(v)_i$ action. Notice that $T^i_\alpha = 0$ implies that $T^i_\beta = 0$ for any fragment β of α starting at some state of α and ending in the last state of α.

Let s' be the first state of α such that no "Collect" message sent by a process $k \neq i$ is present in $\textsc{channel}_{k,j}$ nor in InMsgs_j for any j. State s' exists in α and its time of occurrence is less or equal to $d + \ell$. Indeed, since the execution is nice, all the messages that are in the channels in state s are delivered by time d and messages present in any *InMsgs* set are processed within ℓ time. Since i is the unique leader, in state s' no messages sent by a process $k \neq i$ is present in any channel nor in any *InMsgs* set. Let α' be the fragment of α beginning at s'. Since α' is a fragment of α, we have that α' is nice, process i is the unique leader in α' and $T^i_{\alpha'} = 0$.

If process i has started a round r' by state s' and round r' is successful, then round r' ends by time $T^i_{\alpha'} + 5\ell + 2d = 5\ell + 2d$ in α'. Indeed if the action that brings that system in state s' is a $\mathrm{NewRound}_i$ action for round r' then by Lemma 6.2.24 we have that the round ends by time $T^i_{\alpha'} + 5\ell + 2d = 5\ell + 2d$. If action $\mathrm{NewRound}_i$ for round r' has been executed before, round r' ends even more quickly and the time bound holds anyway. Since the time of occurrence of s' is less or equal to $\ell + d$ we have that round r' ends by time $6\ell + 3d$. Considering the possibility that process i has to wait for an $\mathrm{Init}(v)_i$ action we have that round r' ends by time $T^i_\alpha + 6\ell + 3d$ in α. Hence the lemma is true in this case.

Assume that either (a) process i has started a round r' by state s' but round r is not successful or (b) that process i has not started any round by state s'. In both cases process i executes a $\mathrm{NewRound}_i$ action by time $T^i_{\alpha'} + 7\ell + 2d = 7\ell + 2d$ in α'. Indeed in case (a), by the code of $\textsc{starteralg}_i$, action $\mathrm{CheckRndSuccess}_i$ is executed within $T^i_{\alpha'} + 6\ell + 2d = 6\ell + 2d$ time and it takes additional ℓ time to execute action $\mathrm{NewRound}_i$. In case (b), by the code of $\textsc{bpleader}_i$, action $\mathrm{NewRound}_i$ is executed within ℓ time. Let r'' be the round started by such an action.

Let s'' be the state after the execution of the $\mathrm{NewRound}_i$ action and let α'' be the fragment of α starting in s''. Since α'' is a fragment of α, we have that α'' is nice, process i is the unique leader in α'' and $T^i_{\alpha''} = 0$. We notice that since the time of occurrence of state s' is less or equal to $\ell + d$ the time of occurrence of s'' is less or equal to $8\ell + 3d$ in α.

We now distinguish two possible cases.

CASE 1. Round r'' is successful. In this case, by Lemma 6.2.24 we have that round r'' is successful within $T^i_{\alpha''} + 5\ell + 2d = 5\ell + 2d$ time in α''. Since the time of occurrence of s'' is less or equal to $8\ell + 3d$, we have that round r'' ends by time $13\ell + 5d$ in α. Considering the possibility that process i has to wait for an $\mathrm{Init}(v)_i$ action we have that round r'' ends by time $T^i_\alpha + 13\ell + 5d$ in α. Hence the lemma is true in this case.

CASE 2. Round r'' is not successful.

By the code of STARTERALG$_i$, action NewRound$_i$ is executed within $T^i_{\alpha''} + 7\ell + 2d = 6\ell + 2d$ time in α''. Indeed, it takes $T^i_{\alpha''} + 5\ell + 2d$ to execute action CheckRndSuccess$_i$ and additional ℓ time to execute action NewRound$_i$. Let r''' be the new round started by i with such an action, let s''' be the state of the system after the execution of action NewRound$_i$ and let α''' be the fragment of α'' beginning at s'''. The time of occurrence of s''' is less or equal than $15\ell + 5d$ in α.

Clearly α''' is nice, process i is the unique leader in α'''. Any alive process j that rejected round r'' because of a round $\tilde{r}$, $\tilde{r} > r''$, has responded to the "Collect" message of round r'', with a message $\langle r''$,"OldRound",$\tilde{r}\rangle_{j,i}$ informing the leader i about round $\tilde{r}$. Since α'' is nice all the "OldRound" messages are received before state s'''. Since action NewRound$_i$ uses a round number greater than all the ones received in "OldRound" messages, we have that for any alive process j, $s'''.Commit_j < r'''$. Let $\mathcal{J}$ be the set of alive processes. In state s''', for every $j \in \mathcal{J}$ and any $k \in \mathcal{I}$, CHANNEL$_{k,j}$ does not contain any "Collect" message belonging to any round $\tilde{r} > r'''$ nor such a message is present in any $InMsgs_j$ set (indeed this is true in state s'). Finally since α'' is nice, by definition of nice execution fragment, we have that $\mathcal{J}$ contains a majority of the processes.

Hence we can apply Lemma 6.2.25 to the execution fragment α'''. By Lemma 6.2.25, round r''' is successful within $T^i_{\alpha'''} + 5\ell + 2d = 5\ell + 2d$ time from the beginning of α'''. Since the time of occurrence of s''' is less or equal to $15\ell + 5d$ in α, we have that round r''' ends by time $20\ell + 7d$ in α. Considering the possibility that process i has to wait for an $\text{Init}(v)_i$ action we have that round r''' ends by time $T^i_\alpha + 20\ell + 7d$ in α. Hence the lemma is true also in this case. ■

If the execution is stable for enough time, then the leader election eventually elects a unique leader (Lemma 5.2.3). In the following theorem we consider a nice execution fragment α and we let i be the process eventually elected unique leader. We recall that t^i_α is the time of occurrence of action $\text{Init}(v)_i$ in α and that ℓ and d are constants.

Theorem 6.4.2 *Let α be a nice execution fragment of S_{PAX} starting in a reachable state and lasting for more than $t^i_\alpha + 35\ell + 13d$. Then the leader i executes Decide$(v')_i$ by time $t^i_\alpha + 32\ell + 11d$ from the beginning of α. Moreover by time $t^i_\alpha + 35\ell + 13d$ from the beginning of α any alive process j executes Decide$(v')_j$.*

Proof: Since S_{PAX} contains S_{LEA} and S_{BPX} as subsystems, by Theorem 2.6.10 we can use any property of S_{LEA} and S_{BPX}. Since the execution fragment is nice (and thus stable), by Lemma 5.2.3 there is a unique leader by time $4\ell + 2d$. Let s' be the first unique-leader state of α and let i be the leader. By Lemma 5.2.3 the time of occurrence of s' before or at time $4\ell + 2d$. Let α' be the fragment of α starting in state s'. Since α is nice, α' is nice.

By Lemma 6.4.1 we have that the leader reaches a decision by time $T^i_{\alpha'} + 20\ell + 7d$ from the beginning of α'. Summing up the times and noticing that $T^i_{\alpha'} \leq t^i_{\alpha'} + 7\ell + 2d$ and that $t^i_{\alpha'} \leq t^i_\alpha$ we have that the leader reaches a decision by time $t^i_\alpha + 31\ell + 11d$. Within additional ℓ time action $\text{Decide}(v')_i$ is executed.

The leader reaches a decision by time $t^i_\alpha + 31\ell + 11d$. By Lemma 6.2.20 we have that a decision is reached by every alive process j within additional $3\ell + 2d$ time, that is by time $t^i_\alpha + 34\ell + 13d$. Within additional ℓ time action $\text{Decide}(v')_j$ is executed. ■

6.5 Messages

It is not difficult to see that in a nice execution, which is an execution with no failures, the number of messages spent in a round is linear in the number of processes. Indeed in a successful round the leader broadcasts two messages and the agents respond to the leader's messages. Once the leader reached a decision another broadcast is enough to spread this decision to the agents. It is easy to see that, if everything goes well, at most $6n$ messages are sent to have all the alive processes reach the decision.

However failures may cause the sending of extra messages. It is not difficult to construct situations where the number of messages sent is quadratic in the number of processes. For example if we have that before i becomes the unique leader, all the processes act as leaders and send messages, even if i becomes the unique leader and conducts a successful round, there are $\Theta(n^2)$ messages in the channels which are delivered to the agents which respond to these messages.

Automaton BPSUCCESS keeps sending messages to processes that do not acknowledge the "Success" messages. If a process is dead and never recovers, an infinite number of messages is sent. In a real implementation, clearly the leader should not send messages to dead processes.

Finally the automaton DETECTOR sends an infinite number of messages. However the information provided by this automaton can be used also by other applications.

6.6 Concluding remarks

The PAXOS algorithm was devised in [29]. In this chapter we have provided a new presentation of the PAXOS algorithm. We conclude this chapter with a few remarks.

The first remark concerns the use of majorities for info-quorums and accepting-quorums. The only property that is used is that there exists at least one process common to any info-quorum and any accepting-quorum. Thus any quorum scheme for info-quorums and accepting-quorums that guarantees the above property can be used.

As pointed out in also [31], the amount of stable storage needed can be reduced to a very few state variables. These are the last round started by a leader (which is stored in the *CurRnd* variable), the last round in which an agent accepted the value and the value of that round (variables *LastR*, *LastV*), and the round for which an agent is committed (variable *Commit*). These variables are used to keep consistency, that is, to always propose values that are consistent with previously proposed values, so if they are lost then consistency might not be preserved. In our setting we assumed that the entire state of the processes is in stable storage, but in a practical implementation only the variables described above need to be stable.

A practical implementation of PAXOS should cope with some failures before abandoning a round. For example a message could be sent twice, since duplication is not a problem for the algorithm (it may only affect the message analysis), or the time bound checking may be done later than the earliest possible time to allow some delay in the delivery of messages.

A recover may cause a delay. Indeed if the recovered process has a bigger identifier than the one of the leader then it will become the leader and will start new rounds, possibly preventing the old round from succeeding. As suggested in Lamport's original paper, one could use a different leader election strategy which keeps a leader as long as it does not fail. However it is not clear to us how to design such a strategy.

Chapter 7

The MULTIPAXOS algorithm

The PAXOS algorithm allows processes to reach consensus on one value. We consider now the situation in which consensus has to be reached on a sequence of values; more precisely, for each integer k, processes need to reach consensus on the k-th value. The MULTIPAXOS algorithm reaches consensus on a sequence of values; it was discovered by Lamport at the same time as PAXOS [29].

7.1 Overview

To achieve consensus on a sequence of values we can use an instance of PAXOS for each integer k, so that the k-th instance is used to agree on the k-th value. Since we need an instance of PAXOS to agree on the k-th value, we need for each integer k an instance of the BASICPAXOS and STARTERALG automata. To distinguish instances we use an additional parameter that specifies the ordinal number of the instance. So, we have BASICPAXOS(1), BASICPAXOS(2), BASICPAXOS(3), etc., where BASICPAXOS(k) is used to agree on the k-th value. This additional parameter will be present in each action. For instance, the $\text{Init}(v)_i$ and $\text{Decide}(v')_i$ actions of process i become $\text{Init}(k, v)_i$ and $\text{Decide}(k, v')_i$ in BASICPAXOS$(k)_i$. Similar modifications are needed for all other actions. The STARTERALG$_i$ automaton for process i has to be modified in a similar way. Also, messages belonging to the k-th instance need to be tagged with k.

This simple approach has the problem that an infinite number of instances must be started unless we know in advance how many instances of PAXOS are needed. We have not defined the composition of Clock GTA for an infinite number of automata (see Chapter 2).

In the following section we follow a different approach consisting of modifying the BASICPAXOS and STARTERALG automata of PAXOS to obtain the MULTIPAXOS algorithm. This differs from the approach described above because we do not have separate automata for each single instance. The MULTIPAXOS algorithms takes advantage of the fact that, in a normal situation, there is a unique leader that runs all the instances of PAXOS. The leader can use a single message for step 1 of all the instances. Similarly step 2 can also be handled grouping all the instances together. Then, from step 3 on each instance must proceed separately; however step 3 is performed only when an initial value is provided.

Though the approach described above is conceptually simple, it requires some change to the code of the automata we developed in Chapter 6. To implement MULTIPAXOS we need to modify BASICPAXOS and STARTERALG. Indeed BASICPAXOS and STARTERALG are designed to handle a single instance of PAXOS, while now we need to handle many instances all together for the first two steps of a round. In this section we design two automata similar to BASICPAXOS and STARTERALG that handle multiple instances of PAXOS. We call them MULTIBASICPAXOS and MULTISTARTERALG.

7.2 Automaton MULTIBASICPAXOS.

Automaton MULTIBASICPAXOS has, of course, the same structure as BASICPAXOS, thus goes through the same sequence of steps of a round with the difference that now steps 1 and 2 are executed only once and not repeated by each instance. The remaining steps are handled separately for each instance of PAXOS.

When initiating new rounds MULTIBASICPAXOS uses the same round number for all the instances. This allows the leader to send only one "Collect" message to all the agents and this message serves for all the instances of PAXOS. When responding to a "Collect" message for a round r, agents have to send information about all the instances of PAXOS in which they are involved; for each of them they have to specify the same information as in BASICPAXOS, i.e., the number of the last round in which they accepted the value being proposed and the value of that round. We recall that an agent, by responding to a "Collect" message for a round r, also commits to not accept the value of any round with round number less than r; this commitment is made for all the instances of PAXOS.

Once the leader has executed steps 1 and 2, it is ready to execute step 3 for every instance for which there is an initial value. For instances for which there is no initial value provided, the leader can proceed with step 3 as soon as there will be an initial value.

Next, we give a description of the steps of MULTIBASICPAXOS by relating them to those of BASICPAXOS, so that it is possible to emphasize the differences.

1. To initiate a round, the leader sends a message to all agents specifying the number r of the new round and also the set of instances for which the leader already knows the outcome. This message serves as "Collect" message for all the instances of PAXOS for which a decision has not been reached yet. This is an infinite set, but only for a finite number of instances is there information to exchange. Since agents may be not aware of the outcomes of instances for which the leader has already reached a decision, the leader sends in the "Collect" message, along with the round number, also the instances of PAXOS for which it already knows the decision.

2. An agent that receives a message sent in step 1 from the leader of the round, responds giving its own information about rounds previously conducted for all the instances of PAXOS for which it has information to give to the leader. This information is as in BASICPAXOS, that is, for each instance the agent sends the last round in which it accepted the proposed value and the value of that round. Only for a finite number of instances does the agent have information. The agent makes the same kind of commitment as in BASICPAXOS. That is it commits, in any instance, to not accept the value of any round with round

number less than r. An agent may have already reached a decision for instances for which the leader still does not know the decision. Hence the agent also informs the leader of any decision already made.

3. Once the leader has gathered responses from a majority of the processes it can propose a value for each instance of PAXOS for which it has an initial value. As in BASICPAXOS, it sends a "Begin" message asking to accept that value. For instances for which there is no initial value, the leader does not perform this step. However, as soon as there is an initial value, the leader can perform this step. Notice that step 3 is performed separately for each instance.

4. An agent that receives a message from the leader of the round sent in step 3 of a particular instance, responds by accepting the proposed value if it is not committed for a round with a larger round number.

5. If the leader of a round receives, for a particular instance, "Accept" messages from a majority of processes, then, for that particular instance, a decision is made.

Once the leader has made a decision for a particular instance, it broadcasts that decision as in BASICPAXOS.

It is worth to notice that since steps 1 and 2 are handled with all the instances grouped together, there is a unique info-quorum, while, since from step 3 on each instance proceeds separately, there is an accepting-quorum for each instance (two instances may have different accepting-quorums).

Figures 7-1, 7-2, 7-3, 7-4 and 7-5 show the code fragments of automata BMPLEADER$_i$, BMPAGENT$_i$ and BMPSUCCESS$_i$ for process i. Automaton MULTIBASICPAXOS$_i$ for process i is obtained composing these three automata. In addition to the code fragments, we provide here some comments. The first general comment is that MULTIBASICPAXOS is really similar to BASICPAXOS and the differences are just technicalities due to the fact that MULTIBASICPAXOS handles multiple instances of PAXOS all together for the first two steps of a round. This clearly results in a more complicated code, at least for some parts of the automaton. We refer the reader to the description of the code of BASICPAXOS and in the following we give specific comments on those parts of the automaton that required significant changes. We will follow the same style used for BASICPAXOS by describing the messages used and, for each automaton, the state variables and the actions.

Messages. Messages are as in BASICPAXOS. The structure of the messages is slightly different. The following description of the messages is done assuming that process i is the leader.

1. "Collect" messages, $m = \langle r, \text{"Collect"}, D, W\rangle_{i,j}$. This message is as the "Collect" message of BASICPAXOS$_i$. Moreover, it specifies also the set D of all the instances for which the leader already knows the decision and the set W of instances for which the leader has an initial value but not a decision yet.

2. "Last" messages, $m = \langle r, \text{"Last"}, D', W', \{(k, r_k, v_k) | k \in W'\}\rangle_{j,i}$. As in BASICPAXOS$_i$ an agent responds to a "Collect" message with a "Last" message. The message includes a set D' containing pairs $(k, \mathit{Decision}(k))$ for all the instances for which the agent knows the decision and the leader does not. The message includes also a set W' which contains all the instances of the set W of the "Collect" message plus those instances for which the agent has an initial value while the leader does not. Finally for each instance k in W' the agent sends the round number r_k of the latest accepted round for instance k and the value v_k of round r_k.

3. "Begin" messages, $m = \langle k, r, \text{"Begin"}, v\rangle_{i,j}$. This message is as in BASICPAXOS$_i$ with the difference that the particular instance k to which it is pertinent is specified.

4. "Accept" messages, $m = \langle k, r, \text{"Accept"}\rangle_{j,i}$. This message is as in BASICPAXOS$_i$ with the difference that the particular instance k to which it is pertinent is specified.

BMPLEADER$_i$

Signature:

Input: Receive$(m)_{j,i}$, $m \in \{$"Last", "Accept", "OldRound"$\}$
Init$(k,v)_i$, NewRound$_i$, Stop$_i$, Recover$_i$, Leader$_i$, NotLeader$_i$
Internal: Collect$_i$, GatherLast$_i$, BeginCast$(k)_i$, GatherAccept$(k)_i$, GatherOldRound$_i$
Output: Send$(m)_{i,j}$, $m \in \{$"Collect", "Begin"$\}$
Gathered$(k,v)_i$, Continue$(k)_i$, RndSuccess$(k,v)_i$

States:

Status $\in$ {`alive`,`stopped`}	init. `alive`	*HighestRnd* $\in \mathcal{R}$	init. $(0,i)$
IamLeader, a boolean	init. `false`	*CurRnd* $\in \mathcal{R}$	init. $(0,i)$
Mode $\in$ {`done`,`collect` `gatherlast`}	init. `done`	*Value*, array of $V \cup$ {`nil`}	init. all `nil`
Mode, array of $\in$ {`gathered`, `wait`,`begincast`, `gatheraccept`, `decided`,`done`}	init. all `done`	*ValFrom*, array of $\mathcal{R}$	init. all $(0,i)$
InitValue, array of $V \cup$ `nil`	init. all `nil`	*InfoQuo* $\in 2^{\mathcal{I}}$	init. $\{\}$
Decision, array of $V \cup$ {`nil`}	init. all `nil`	*AcceptQuo*, array of $2^{\mathcal{I}}$	init. all $\{\}$
		InMsgs, multiset of msgs	init. $\{\}$
		OutMsgs, multiset of msgs	init. $\{\}$

Derived Variable:

LeaderAlive, a boolean, `true` iff *Status* = `alive` and *IamLeader* = `true`

Actions:

input Stop$_i$
Eff: *Status* := `stopped`

input Leader$_i$
Eff: if *Status* = `alive` then
IamLeader := `true`

output Send$(m)_{i,j}$
Pre: *Status* = `alive`
$m_{i,j} \in$ *OutMsgs*
Eff: remove $m_{i,j}$ from *OutMsgs*

input NewRound$_i$
Eff: if *LeaderAlive* = `true` then
CurRnd := *HighestRnd* $+_i$ 1
HighestRnd := *CurRnd*
Mode := `collect`
$\forall k$, *Mode*(k) := `done`

input Recover$_i$
Eff: *Status* := `alive`

input NotLeader$_i$
Eff: if *Status* = `alive` then
IamLeader := `false`

input Receive$(m)_{j,i}$
Eff: if *Status* = `alive` then
add $m_{j,i}$ to *InMsgs*

input Init$_i(k,v)$
Eff: if *Status* = `alive` then
InitValue(k) := v

Tasks and bounds:

{Collect$_i$, Gathered$(k,v)_i$, Continue$(k)_i$, BeginCast$(k)_i$, RndSuccess$(k,v)_i$: $k \in \mathbb{N}$}, bounds $[0,\ell]$
{GatherLast$(k,m)_i$: $k \in \mathbb{N}$, $m \in$ *InMsgs*, m is a "Last" message }, bounds $[0,\ell]$
{GatherAccept$(k,m)_i$: $k \in \mathbb{N}$, $m \in$ *InMsgs*, m is an "Accept" message }, bounds $[0,\ell]$
{GatherOldRound$(m)_i$: $m \in$ *InMsgs*, m is an "OldRound" message }, bounds $[0,\ell]$
{Send$(m)_i m \in \mathcal{M}$}, bounds $[0,\ell]$

Figure 7-1: Automaton BMPLEADER for process i (part 1)

Actions:

internal Collect$_i$
Pre: $LeaderAlive = \texttt{true}$
$Mode = \texttt{collect}$
Eff: $InfoQuo := \{\}$
$D := \{k | Decision(k) \neq \texttt{nil}\}$
$\forall k \notin D$
$Value(k) := InitValue(k)$
$ValFrom(k) := (0, i)$
$AcceptQuo(k) := \{\}$
$W := \{k |\ InitValue(k) \neq \texttt{nil}$ and
$Decision(k) = \texttt{nil}\}$
$\forall j$ put $\langle CurRnd, \text{“Collect”}, D, W\rangle_{i,j}$
in $OutMsgs$
$Mode := \texttt{gatherlast}$

internal GatherLast$_i$
Pre: $LeaderAlive = \texttt{true}$
$Mode = \texttt{gatherlast}$
$m = \langle r\ , \text{“Last”}, D, W,$
$\{(k, r_k, v_k) | k \in W\}\rangle_{j,i} \in InMsgs$
$CurRnd = r$
Eff: remove m from $InMsgs$
$\forall (k, v) \in D$ do
$Decision(k) := v$
$Mode(k) := \texttt{decided}$
$InfoQuo := InfoQuo \cup \{j\}$
$\forall k \in W$ do
if $ValFrom(k) < r_k$
and $v_k \neq \texttt{nil}$ then
$Value(k) := v_k$
$ValFrom(k) := r_k$
if $|InfoQuo| > n/2$ then
$\forall k \in W,\ Mode(k) := \texttt{gathered}$

output Gathered(k, $Value(k)$)$_i$
Pre: $LeaderAlive = \texttt{true}$
$Mode(k) = \texttt{gathered}$
Eff: $\forall k$ do
if $Value(k) = \texttt{nil}$ and
$InitValue(k) \neq \texttt{nil}$ then
$Value(k) := InitValue(k)$
$Value(k) \neq \texttt{nil}$ then
$Mode(k) := \texttt{begincast}$
else
$Mode(k) := \texttt{wait}$

output Continue(k)$_i$
Pre: $LeaderAlive = \texttt{true}$
$Mode(k) = \texttt{wait}$
$Value(k) = \texttt{nil}$
$InitValue(k) \neq \texttt{nil}$
Eff: $Value(k) := InitValue(k)$
$Mode(k) := \texttt{begincast}$

internal BeginCast(k)$_i$
Pre: $LeaderAlive = \texttt{true}$
$Mode(k) = \texttt{begincast}$
Eff: $\forall j$ put
$\langle k, CurRnd, \text{“Begin”}, Value(k)\rangle_{i,j}$
in $OutMsgs$
$Mode(k) := \texttt{gatheraccept}$

internal GatherAccept(k)$_i$
Pre: $LeaderAlive = \texttt{true}$
$Mode(k) := \texttt{gatheraccept}$
$m = \langle k, r, \text{“Accept”}\rangle_{j,i} \in InMsgs$
$CurRnd = r$
Eff: remove m from $InMsgs$
$AcceptQuo(k) := AcceptQuo(k) \cup \{j\}$
if $|AcceptQuo(k)| > n/2$ then
$Decision(k) := Value(k)$
$Mode(k) := \texttt{decided}$

output RndSuccess(k,$Decision$)$_i$
Pre: $LeaderAlive = \texttt{true}$
$Mode(k) = \texttt{decided}$
Eff: $Mode(k) = \texttt{done}$

internal GatherOldRound$_i$
Pre: $LeaderAlive = \texttt{true}$
$m = \langle r, \text{“OldRound”}, r'\rangle_{j,i} \in InMsgs$
$CurRnd < r$
Eff: remove m from $InMsgs$
$HighestRnd := r'$

Figure 7-2: Automaton BMPLEADER for process i (part 2)

BMPAGENT$_i$

Signature:

Input: Receive$(m)_{j,i}$, $m \in \{$"Collect", "Begin"$\}$
Init$(k,v)_i$, Stop$_i$, Recover$_i$

Internal: LastAccept$(m)_i$, Accept$(k,m)_i$

Output: Send$(m)_{i,j}$, $m \in \{$"Last", "Accept", "OldRound"$\}$

States:

Status $\in$ {`alive`,`stopped`}	init. `alive`	*Commit* $\in \mathcal{R}$	init. $(0,i)$
LastR, array of $\mathcal{R}$	init. all $(0,i)$	*InMsgs*, multiset of messages	init. $\{\}$
LastV, array of $V \cup \{$`nil`$\}$	init. all `nil`	*OutMsgs*, multiset of messages	init. $\{\}$

Actions:

input Stop$_i$
Eff: *Status* := `stopped`

input Recover$_i$
Eff: *Status* := `alive`

internal LastAccept$(m)_i$
Pre: *Status* = `alive`
$m = \langle r,$"Collect"$,D,W\rangle_{j,i}$
$m \in$ *InMsgs*
Eff: remove m from *InMsgs*
if $r \geq$ *Commit* then
Commit := r
$W'' := \{k \in \mathbb{N} | LastV(k) \neq$ `nil`,
$Decision(k) =$ `nil`$\}$
$W' := W'' \cup W$
$D' := \{(k, Decision(k) \mid$
$k \notin D$, $Decision(k) \neq$ `nil`$\}$
put $\langle r,$"Last"$,D',W',$
$\{(k,r_k,v_k) | k \in W'\}\rangle_{i,j}$
in *OutMsgs*
where $r_k = LastR(k)$
$v_k = LastV(k)$
else
put $\langle r,$"OldRound"$,Commit\rangle_{i,j}$
in *OutMsgs*

output Send$(m)_{i,j}$
Pre: *Status* = `alive`
$m_{i,j} \in$ *OutMsgs*
Eff: remove $m_{i,j}$ from *OutMsgs*

input Receive$(m)_{j,i}$
Eff: if *Status* = `alive` then
add $m_{j,i}$ to *InMsgs*

internal Accept$(k,m)_i$
Pre: *Status* = `alive`
$m = \langle k,r,$"Begin"$,v\rangle_{j,i} \in$ *InMsgs*
Eff: remove m from *InMsgs*
if $r \geq$ *Commit* then
put $\langle k,r,$"Accept"$\rangle_{i,j}$
in *InMsgs*
$LastR(k) := r$
$LastV(k) := v$
else
put $\langle r,$"OldRound"$,Commit\rangle_{i,j}$
in *OutMsgs*

input Init$(k,v)_i$
Eff: if *Status* = `alive` then
if $LastV(k) =$ `nil` then $LastV(k) := v$

Tasks and bounds:

$\{$LastAccept$(m)_i$, $m \in$ *InMsgs*, m is a "Collect" message $\}$, bounds $[0,\ell]$
$\{$Accept$(k,m)_i : k \in \mathbb{N}$, $m \in$ *InMsgs*, m is a "Collect" message $\}$, bounds $[0,\ell]$
$\{$Send$(m)_i : m \in$ *OutMsgs*$\}$, bounds $[0,\ell]$

Figure 7-3: Automaton BMPAGENT for process i

BMPSUCCESS$_i$

Signature:

Input: Receive$(m)_{j,i}$, $m \in \{$"Ack", "Success"$\}$
Stop$_i$, Recover$_i$, Leader$_i$, NotLeader$_i$, RndSuccess$(k, v)_i$
Internal: SendSuccess$_i$, Check$_i$
Output: Send$(m)_{i,j}$, $m \in \{$"Ack", "Success"$\}$
Decide$(v)_i$
Time-passage: $\nu(t)$

State:

$Clock \in \mathbb{R}$	init. arbitrary
$Status \in \{\texttt{alive}, \texttt{stopped}\}$	init. `alive`
IamLeader, a boolean	init. `false`
Decision, array of $V \cup \{\texttt{nil}\}$	init. all `nil`
Prevsend, array of $\mathbb{R} \cup \{\texttt{nil}\}$	init. all `nil`
LastCheck, array of $\mathbb{R} \cup \{\infty\}$	init. all ∞
LastSS, array of $\mathbb{R} \cup \{\infty\}$	init. all ∞
For each $j \in \mathcal{I}$, $k \in \mathbb{N}$	
$Acked(k, j)$, a boolean	init. `false`
$LastSendAck(k, j) \in \mathbb{R} \cup \{\infty\}$	init. ∞
$LastSendSuc(k, j) \in \mathbb{R} \cup \{\infty\}$	init. ∞
$OutAckMsgs(k, j)$, set of msgs	init. $\{\}$
$OutSucMsgs(k, j)$, set of msgs	init. $\{\}$

Actions:

input Stop$_i$
Eff: $Status :=$ `stopped`

input Leader$_i$
Eff: if $Status =$ `alive` and
$IamLeader =$ `false` then
$IamLeader :=$ `true`
$\forall k$, if $Decision(k) \neq$ `nil` then
$LastSS(k) := clock + \ell$
$PrevSend(k) :=$ `nil`

output Send$(m)_{i,j}$
Pre: $Status =$ `alive`
$\exists k$, $m_{i,j} \in OutAckMsgs(k, j)$
Eff: $OutAckMsgs(k, j) := \{\}$
$LastSendAck(k, j) := \infty$

output Send$(m)_{i,j}$
Pre: $Status =$ `alive`
$\exists k$, $m_{i,j} \in OutSucMsgs(k, j)$
Eff: $OutSucMsgs(k, j) := \{\}$
$LastSendSuc(k, j) := \infty$

input Recover$_i$
Eff: $Status :=$ `alive`

input NotLeader$_i$
Eff: if $Status =$ `alive` then
$IamLeader :=$ `false`
$\forall k$ do
$LastSS(k) := \infty$
$LastCheck(k) := \infty$
$\forall j \in \mathcal{I}$
$LastSendSuc(k, j) := \infty$

input Receive$(\langle k, \text{"Ack"}\rangle)_{j,i}$
Eff: if $Status =$ `alive` then
$Acked(k, j) :=$ `true`

input Receive$(\langle k, \text{"Success"}, v\rangle)_{j,i}$
Eff: if $Status =$ `alive` then
$Decision(k) := v$
put $\langle k, \text{"Ack"}\rangle_{i,j}$
into $OutAckMsgs(k, j)$
$LastSendAck(k, j) := Clock + \ell$

Figure 7-4: Automaton BMPSUCCESS for process i (part 1)

input RndSuccess$(k,v)_i$
Eff: if *Status* $=$ `alive` then
    $\mathit{Decision}(k) := v$
    if $\mathit{IamLeader} =$ `true` then
      $\mathit{LastSS}(k) := \mathit{Clock} + \ell$
      $\mathit{PrevSend}(k) :=$ `nil`

internal SendSuccess$_i$
Pre: $\mathit{Status} =$ `alive`
    $\mathit{IamLeader} =$ `true`
    $\exists k$ such that
      $\mathit{Decision}(k) \neq$ `nil`
      $\mathit{PrevSend}(k) =$ `nil`
      $\exists j \neq i,\ \mathit{Acked}(k,j) =$ `false`
Eff: $\forall j \neq i$ such that $\mathit{Acked}(k,j) =$ `false`
      put $\langle$k,"Success",$\mathit{Decision}\rangle_{i,j}$
        in $\mathit{OutSucMsgs}(k,j)$
      $\mathit{LastSendSuc}(k,j) := \mathit{Clock} + \ell$
    $\mathit{PrevSend}(k) := \mathit{Clock}$
    $\mathit{LastCheck}(k) := \mathit{Clock} + (2\ell + 2d) + \ell$
    $\mathit{LastSS}(k) := \infty$

internal Check$_i$
Pre: $\mathit{Status} =$ `alive`
    $\exists k$ such that
      $\mathit{PrevSend}(k) \neq$ `nil`
      $\mathit{Clock} > \mathit{PrevSend}(k) + (2\ell + 2d)$
Eff: $\mathit{PrevSend}(k) :=$ `nil`
    $\mathit{LastSS}(k) := \mathit{Clock} + \ell$
    $\mathit{LastCheck}(k) := \infty$

output Decide$(k,v)_i$
Pre: $\mathit{Status} =$ `alive`
    $\mathit{Decision}(k) \neq$ `nil`
    $\mathit{Decision}(k) = v$
Eff: none

time-passage $\nu(t)$
Pre: $\mathit{Status} =$ `alive`
Eff: Let t' be such that for all k
      $\mathit{Clock}{+}t' \leq \mathit{LastCheck}(k)$
      $\mathit{Clock}{+}t' \leq \mathit{LastSS}(k)$
      and for each $j \in \mathcal{I}$
        $\mathit{Clock}{+}t' \leq \mathit{LastSendAck}(k,j)$
        $\mathit{Clock}{+}t' \leq \mathit{LastSendSuc}(k,j)$
    $\mathit{Clock} := \mathit{Clock}{+}t'$

Figure 7-5: Automaton BMPSUCCESS for process i (part 2)

5. "OldRound" messages, $m = \langle r,$"OldRound"$,r'\rangle_{j,i}$. This message is as in BASICPAXOS$_i$. Notice that there is no need to specify any instance since when a new round is started, it is started for all the instances.

6. "Success" messages, $m = \langle k,$"Success"$,v\rangle_{i,j}$. This message is as in BASICPAXOS$_i$ with the difference that the particular instance k to which it is pertinent is specified.

7. "Ack" messages, $m = \langle k,$"Ack"$\rangle_{j,i}$. This message is as in BASICPAXOS$_i$ with the difference that the particular instance k to which it is pertinent is specified.

Most state variables and automaton actions are similar to the correspondent state variables and automaton actions of BASICPAXOS. We will only describe those state variables and automata actions that required significant changes. For variables we need to use arrays indexed by the instance number. Most of the actions are as in BASICPAXOS$_i$ with the difference that a parameter k specifying the instance is present.

This is true especially for actions relative to steps 3, 4, and 5 and for BMPSUCCESS$_i$. Actions relative to steps 1 and 2 needed major rewriting since in MULTIBASICPAXOS$_i$ they handle multiple instances of PAXOS all together.

Automaton BMPLEADER$_i$**.** Variables *InitValue*, *Decision*, *Value*, *ValFrom* and *AcceptQuo* are now arrays of variables indexed by the instance number: we need the information stored in these variables for each instance. Variable *HighestRnd*, *CurRnd* and *InfoQuo* are not arrays because there is always one current round number and only one info-quorum (used for all the instances). Variable *Mode* deserves some more comments: in BMPLEADER$_i$ we have a scalar variable *Mode* which is used for the first two steps, then, since from the third step on each instance is run separately, we have another variable *Mode* which is an array. Notice that values `collect` and `gatherlast` of variable *Mode* are relative to the first two steps of a round and that values `gathered, wait, begincast, gatheraccept, decided` are relative to the other steps of a round. Value `done` is used either when no round has been started yet and also when a round has been completed.

Action Collect$_i$ first computes the set D of PAXOS instances for which a decision is already known. Then initializes the state variables pertinent to all the potential instances of PAXOS, which are all the ones not included in D. Notice that even though this is potentially an infinite set, we need to initialize those variables only for a finite number of instances. Then it computes the set W of instances of PAXOS for which the leader has an initial value but not yet a decision. Finally a "Collect" message is sent to all the agents. Action GatherLast$_i$ takes care of the receipt of the responses to the "Collect" message. It processes "Last" messages by updating, as BASICPAXOS$_i$ does, the state variables pertinent to all the instances for which information is contained in the "Last" message. Also if the agent is informing the leader of a decision of which the leader is not aware, then the leader immediately sets its *Decision* variable. When a "Last" message is received from a majority of the processes, the info-quorum is fixed. At this point, each instance for which there is an initial value can go on with step 3 of the round. Action Continue$_i$ takes care of those instances for which after the info-quorum is fixed by the GatherLast_i action, there is no initial value. Once the info-quorum has been fixed, action $\text{Gathered}(k, v)$ is executed and if a value for the current round and a particular instance k is already available then action $\text{BeginCast}(k)_i$ is enabled, otherwise action $\text{Continue}(k)_i$ enables $\text{BeginCast}(k)_i$ as soon as there is an initial value. Other actions are similar to the corresponding actions in BPLEADER_i.

Automaton BMPAGENT_i. Variables *LastR* and *LastV* are now arrays of variables indexed by the instance number, while variable *Commit* is a scalar variable; indeed there is always only one round number used for all the instances.

Action LastAccept_i responds to the "Collect" message. If the agent is not committed for the round number specified in the "Collect" message it commits for that round and sends to the leader the following information: the set D' of PAXOS instances for which the agent knows the decision while the leader does not, and for each of such instances, also the decision; for each instance in the set W of the "Collect" message and also for each instance for which the agent has an initial value while the leader does not, the usual information, about the last round in which the process accepted the value of the round and the value of that round, is included in the message. Action $\text{Accept}(k)_i$ and $\text{Init}(k, v)_i$ are similar to the corresponding actions in BPAGENT_i.

Automaton BMPSUCCESS_i. This automaton is very similar to BPSUCCESS_i. The only difference is that now the leader sends a "Success" message for any instance for which there is a decision and there are agents that have not sent an acknowledgment.

7.3 Automaton MULTISTARTERALG

As for BASICPAXOS, also for MULTIBASICPAXOS we need an automaton that takes care of starting new rounds when necessary, i.e., when a decision is not reached within some time bound. We call this automaton MULTISTARTERALG. The task of MULTISTARTERALG is the same as the one of STARTERALG: it has to check that rounds are successful within the expected time bounds. This time bound checking is done separately for each instance.

Figures 7-6 and 7-7 show automaton MULTISTARTERALG$_i$ for process i. The automaton is similar to automaton STARTERALG$_i$. The difference is that the time bound checking is done, separately, for each instance. A new round is started if there is an instance for which actions Gathered$_i$ and RndSuccess$_i$ are not executed within the expected time bounds.

7.4 Correctness and analysis

We do not prove formally the correctness of the code provided in this section. However the correctness follows from the correctness of PAXOS. Indeed for every instance of PAXOS, the code of MULTIPAXOS provided in this section does exactly the same thing that PAXOS does; the only difference is that step 1 (as well as step 2) is handled in a single shot for all the instances. It follows that Theorem 6.4.2 can be restated for each instance k of PAXOS. In the following theorem we consider a nice execution fragment α and we assume that i is eventually elected leader (by Lemma 5.2.3 this happens by time $4\ell + 2d$ in α).

In the following theorem $t^i_\alpha(k)$ denotes t^i_α for instance k. The formal definition of $t^i_\alpha(k)$ is obtained from the definition of t^i_α (see Definition 6.2.21) by changing $\text{Init}(v)_i$ in $\text{Init}(k,v)_i$.

Theorem 7.4.1 *Let α be a nice execution fragment of S_{MPX} starting in a reachable state and lasting for more than $t^i_\alpha(k) + 35\ell + 13d$. Then the leader i executes Decide$(k,v')_i$ by time $t^i_\alpha(k) + 35\ell + 11d$ from the beginning of α. Moreover by time $t^i_\alpha(k) + 32\ell + 13d$ from the beginning of α any alive process j executes Decide$(k,v')_j$.*

7.5 Concluding remarks

In this chapter we have described the MULTIPAXOS protocol. MULTIPAXOS is a variation of the PAXOS algorithm. It was discovered by Lamport at the same time as PAXOS [29].

MULTISTARTERALG$_i$

Signature:

Input:	Leader$_i$, NotLeader$_i$, Stop$_i$, Recover$_i$, Gathered$(v)_i$, Continue$_i$, RndSuccess$(v)_i$
Internal:	CheckGathered$_i$, CheckRndSuccess$_i$
Output:	NewRound$_i$
Time-passage:	$\nu(t)$

State:

$Clock \in \mathbb{R}$	init. arbitrary	$DlineSuc$, array of $\mathbb{R} \cup \{\infty\}$	init. all `nil`
$Status \in \{$`alive`, `stopped`$\}$	init. `alive`	$DlineGat$, array of $\mathbb{R} \cup \{\infty\}$	init. all `nil`
$IamLeader$, a boolean	init. `false`	$LastNR \in \mathbb{R} \cup \{\infty\}$	init. ∞
$Start$, a boolean	init. `false`	$RndSuccess$, array of boolean	init. all`false`

Actions:

input Stop$_i$
  Eff: $Status :=$ `stopped`

input Recover$_i$
  Eff: $Status :=$ `alive`

input Leader$_i$
  Eff: if $Status =$ `alive` then
      if $IamLeader =$ `false` then
        $IamLeader :=$ `true`
        if exists k such that
        $RndSuccess(k) =$ `false` then
          $Start :=$ `true`
          $LastNR := Clock + \ell$
          $\forall k'$ do
            $DlineGat(k') := \infty$
            $DlineSuc(k') := \infty$

input NotLeader$_i$
  Eff: if $Status =$ `alive` then
      $LastNR := \infty$
      $IamLeader :=$ `false`
      $\forall k$ do
        $DlineSus := \infty$
        $DlineGat := \infty$

input Continue$(k)_i$
  Eff: if $Status =$ `alive` then
      $DlineSuc(k) := Clock + 6\ell + 2d$

output NewRound$_i$
  Pre: $Status =$ `alive`
    $IamLeader =$ `true`
    $Start =$ `true`
  Eff: $Start :=$ `false`
    $LastNR := \infty$
    $\forall k$ do
      $DlineGat(k) := Clock + 6\ell + 2d$
      $DlineSuc(k) :=$ `nil`

input Gathered$(k, v)_i$
  Eff: if $Status =$ `alive` then
      $DlineGat(k) := \infty$
      if $v \neq$ `nil` then
        $DlineSuc(k) := Clock + 6\ell + 2d$

Figure 7-6: Automaton MULTISTARTERALG for process i (part 1)

input RndSuccess$(k,v)_i$
Eff: if *Status* = `alive` then
$\quad$ *RndSuccess*(k) := `true`
$\quad$ *DlineGat*$(k) := \infty$
$\quad$ *DlineSuc*$(k) := \infty$
$\quad$ *LastNR* $:= \infty$

internal CheckGathered$_i$
Pre: *Status* = `alive`
$\quad$ *IamLeader* = `true`
$\quad$ $\exists k$ such that
$\quad\quad$ *DlineGat*$(k) \neq$ `nil`
$\quad\quad$ *Clock* $>$ *DlineGat*(k)
Eff: *DlineGat*$(k) := \infty$
$\quad$ *Start* := `true`
$\quad$ *LastNR* := *Clock* $+ \ell$

internal CheckRndSuccess$_i$
Pre: *Status* = `alive`
$\quad$ *IamLeader* = `true`
$\quad$ $\exists k$ such that
$\quad\quad$ *DlineSuc*$(k) \neq$ `nil`
$\quad\quad$ *Clock* $>$ *DlineSuc*(k)
Eff: *DlineSuc*$(k) := \infty$
$\quad$ *Start* := `true`
$\quad$ *LastNR* := *Clock* $+ \ell$

time-passage $\nu(t)$
Pre: *Status* = `alive`
Eff: Let t' be such that for all k
$\quad\quad$ *Clock* $+ t' \leq$ *DlineGat*$(k) + \ell$
$\quad\quad$ *Clock* $+ t' \leq$ *DlineSuc*$(k) + \ell$
$\quad\quad$ *Clock* $+ t' \leq$ *LastNR*
$\quad$ *Clock* := *Clock* $+ t'$

Figure 7-7: Automaton MULTISTARTERALG for process i (part 2)

MULTIPAXOS achieves consensus on a sequence of values utilizing an instance of PAXOS for each of them. MULTIPAXOS uses an instance of PAXOS to agree on each value of the sequence; remarks about PAXOS provided at the end of Chapter 6 apply also for MULTIPAXOS. We refer the reader to those remarks.

Chapter 8

Application to data replication

In this chapter we show how to use MULTIPAXOS to implement a data replication algorithm.

8.1 Overview

Providing distributed and concurrent access to data objects is an important issue in distributed computing. The simplest implementation maintains the object at a single process which is accessed by multiple clients. However this approach does not scale well as the number of clients increases and it is not fault-tolerant. Data replication allows faster access and provides fault tolerance by replicating the data object at several processes.

One of the best known replication techniques is *majority voting* (e.g., [20, 23]). With this technique both update (write) and non-update (read) operations are performed at a majority of the processes of the distributed system. This scheme can be extended to consider any "write quorum" for an update operation and any "read quorum" for a non-update operation. Write quorums and read quorums are just sets of processes satisfying the property that any two quorums, one of which is a write quorum and the other one is a read quorum, intersect (e.g., [16]). A simple quorum scheme is the write-all/read-one scheme (e.g., [6]) which gives fast access for non-update operations.

Another well-known replication technique relies on a *primary* copy. A distinguished process is considered the primary copy and it coordinates the computation: the clients request operations of the primary copy and the primary copy decides which other copies must be involved in performing the operation. The primary copy technique works better in practice if the primary copy does not fail. Complex recovery mechanisms are needed when the primary copy crashes. Various data replication algorithms based on the primary copy technique have been devised (e.g., [13, 14, 34]).

Replication of the data object raises the issue of consistency among the replicas. These consistency issues depend on what requirements the replicated data has to satisfy. The strongest possible of such requirements is *atomicity*: clients accessing the replicated object obtain results as if there was a unique copy. Primary copy algorithms [1, 34] and voting algorithms [20, 23] are used to achieve atomicity. Achieving atomicity is expensive; therefore weaker consistency requirements are also considered. One of these weaker consistency requirements is *sequential consistency* [26], which allows operations to be re-ordered as long as they remain consistent with the view of individual clients. We remark that, though weaker than atomicity, sequential consistency is a strong consistency requirement.

8.2 Sequential consistency

In this section we formally define a sequential consistent read/update object. Sequential consistency has been first defined by Lamport [26]. We base our definition on the one given in [15] which relies on the notion of atomic object [27, 28] (see also [35] for a description of an atomic object).

Formally a read/update shared object is defined by the set $\mathcal{O}$ of the possible states that the object can assume, a distinguished initial state O_0, and set $\mathcal{U}$ of update operations which are functions $up : \mathcal{O} \rightarrow \mathcal{O}$.

We assume that for each process i of the distributed system implementing the read/update shared object, there is a client i and that client i interacts only with process i. The interface between the object and the clients consists of request actions and report actions. In particular the client i requests a read by executing action Request-read$_i$ and receives a report to the read request when Report-read$(O)_i$ is executed; similarly a client i requests an update operation by executing action Request-update$(up)_i$ and receives the report when action Report-update$_i$ is executed.

If β is a sequence of actions, we denote by $\beta|i$ the subsequence of β consisting of Request-read$_i$, Report-read$(O)_i$, Request-update$(up)_i$ and Report-update$_i$. This subsequence represents the interactions between client i and the read/update shared object.

We will only consider *client-well-formed* sequence of actions β for which $\beta|i$, for every client i, does not contain two request events without an intervening report, i.e., we assume that a client does not request a new operation before receiving the report of the previous request. A sequence of action β is *complete* if for every request event there is a corresponding report event. If β is a complete client-well-formed sequence of actions, we define the *totally-precedes* partial order on the operations that occur in β as follows: an operation o_1 *totally-precedes* an operation o_2 if the report event of operation o_1 occurs before the request event of operation o_2.

In an atomic object, the operations appear "as if" they happened in some sequential order. The idea of "atomic object" originated in [27, 28]. Here we use the formal definition given in Chapter 13 of [35]. In a sequentially consistent object the above atomic requirement is weakened by allowing events to be reordered as long as the view of each client i does not change. Formally a sequence β of request/report actions is *sequentially consistent* if there exists an atomic sequence γ such that $\gamma|i = \beta|i$, for each client i. That is, a sequentially consistent sequence "looks like" an atomic sequence to each individual client, even though the sequence may not be atomic. A read/update shared object is *sequentially consistent* if all the possible sequence of request/report actions are sequentially consistent.

8.3 Using MULTIPAXOS

In this section we will see how to use MULTIPAXOS to design a data replication algorithm that guarantees sequential consistency and provides the same fault tolerance and liveness properties as MULTIPAXOS. The resulting algorithm lies between the two replication techniques discussed at the beginning of the chapter. It is similar to voting schemes since it uses majorities to achieve consistency and it is similar to primary copy techniques since a unique leader is required to achieve termination. Using MULTIPAXOS gives much flexibility. For instance, it is not a disaster when there are two or more "primary" copies. This can only slow down the computation, but never results in inconsistencies. The high fault tolerance of MULTIPAXOS results in a highly fault tolerant data replication algorithm, i.e., process stop and recovery, loss, duplication and reordering of messages, timing failures are tolerated. However, though operations are installed, it is possible that a particular requested operation is never installed.

We can use MULTIPAXOS in the following way. Each process in the system maintains a copy of the data object. When client i requests an update operation, process i proposes that operation in an instance of MULTIPAXOS. When an update operation is the output value of an instance of MULTIPAXOS and the previous update has been applied, a process updates its local copy and the process that received the request for the update gives back a report to its client. A read request can be immediately satisfied returning the current state of the local copy.

It is clear that the use of MULTIPAXOS gives consistency across the whole sequence $up_1, up_2, up_3, ...$ of update operations, since each operation is agreed upon by all the processes. In order for a process to be able to apply operation up_k, the process must first apply operation up_{k-1}. Hence it is necessary that there be no gaps in the sequence of update operations. A gap is an integer k for which processes never reach a decision on the k-th update. This is possible, for example, if no process proposes an update operation as the k-th one. Though making sure that the sequence of update operations does not contain a gap enables the processes to always apply new operations, it is possible to have a kind of "starvation" in which a requested update operation never gets satisfied because other updates are requested and satisfied. We will discuss this in more detail later.

8.3.1 The code

Figures 8-1 and 8-2 show the code of automaton DATAREPLICATION$_i$ for process i. This automaton implements a data replication algorithm using MULTIPAXOS as a subroutine. It accepts requests from a client; read requests are immediately satisfied by returning the current state of the local copy of the object while update requests need to be agreed upon by all the processes and thus an update operation is proposed in the various instances of PAXOS until the operation is the outcome of an instance of PAXOS. When the requested operation is the outcome of a particular instance k of MULTIPAXOS and the $(k-1)$-th update operation has been applied to the object, then the k-th update operation can be applied to the object and a report can be given back to the client that requested the update operation.

Figure 8-3 shows the interactions between the DATAREPLICATION automaton and MULTIPAXOS and also the interactions between the DATAREPLICATION automaton and the clients.

To distinguish operations requested by different clients we pair each operation up with the identifier of the client requesting the update operation. Thus the set V of possible initial values for the instances of PAXOS is the set of pairs (up, i), where up is an operation on the object O and $i \in \mathcal{I}$ is a process identifier.

Next we provide some comments about the code of automaton DATAREPLICATION$_i$.

Automaton actions. Actions Request-update$(up)_i$, Request-read$_i$, Report-update$_i$ and Report-read$(O)_i$ constitute the interface to the client. A client requests an update operation up by executing action Request-update$(up)_i$ and gets back the result r when action Report-update$(r)_i$ is executed by the DATAREPLICATION$_i$ automaton. Similarly a client requests a read operation by executing action Request-read$_i$ and gets back the status of the object O when action Report-read$(O)_i$ is executed by the DATAREPLICATION$_i$ automaton.

DATAREPLICATION$_i$

Signature:

Input:	Receive$(m)_{j,i}$, Decide$(k,v)_i$, Request-update$(up)_i$, Request-read$_i$
Internal:	SendWantPaxos$_i$, RecWantPaxos$_i$, Update$_i$, RePropose$(k)_i$
Output:	Send$(m)_{i,j}$, Init$(k,v)_i$, Report-update$_i$, Report-read$(O)_i$

States:

Propose, array of $V \cup \{\texttt{nil}\}$,	initially `nil` everywhere
Decision, array of $V \cup \{\texttt{nil}\}$,	initially `nil` everywhere
S, an integer,	initially 1
X, a pair (O,k) with $O \in \mathcal{O}$, $k \in \mathbb{N}$	initially $(O_0, 0)$
CurRead $\in \mathcal{O} \cup \{\texttt{nil}\}$,	initially `nil`
Proposed, array of booleans,	initially `false` everywhere
Reproposed, array of booleans,	initially `false` everywhere
InMsgs, multiset of messages,	initially $\{\}$
OutMsgs, multiset of messages,	initially $\{\}$

Tasks and bounds:

$\{$Init$_i\}$, bounds $[0,\ell]$
$\{$RecWantPaxos$_i\}$, bounds $[0,\ell]$
$\{$SendWantPaxos$_i\}$, bounds $[0,\ell]$
$\{$Report-update$_i$, Update$_i\}$, bounds $[0,\ell]$
$\{$Report-read$(O)_i\}$, bounds $[0,\ell]$
$\{$RePropose$(k)_i\}$, bounds $[0,\ell]$
$\{$Send$(m)_{i,j} : m \in \mathcal{M}\}$, bounds $[0,\ell]$

Figure 8-1: Automaton DATAREPLICATION for process i (part 1)

Actions:

output Send$(m)_{i,j}$
Pre: $m_{i,j} \in \mathit{OutMsgs}$
Eff: remove $m_{i,j}$ from $\mathit{OutMsgs}$

input Receive$(m)_{j,i}$
Eff: add m to InMsgs

input Request-read$_i$
Eff: $\mathit{CurRead} := O$, where $X = (O, k)$

output Report-read$(O)_i$
Pre: $\mathit{CurRead} = O$
Eff: $\mathit{CurRead} := \texttt{nil}$

input Request-update$(up)_i$
Eff: $\mathit{Propose}(S) := (up, i)$
$S := S + 1$

output Init$_i(k, (up, j))$
Pre: $\mathit{Propose}(k) = (up, j)$
$\mathit{Proposed}(k) = \texttt{false}$
$\mathit{Decision}(k) = \texttt{nil}$
Eff: $\mathit{Proposed}(k) := \texttt{true}$

internal SendWantPaxos$_i$
Pre: $\mathit{Propose}(k) = (up, i)$
$\mathit{Decision}(k) = \texttt{nil}$
Eff: $\forall j$ put (“WantPaxos”,S,$(up, i))_{i,j}$ in $\mathit{OutMsgs}$

internal RecWantPaxos$_i$
Pre: m=(“WantPaxos”,k,(up, j)) in InMsgs
Eff: remove m from InMsgs
if $\mathit{Propose}(k) = \texttt{nil}$ then
$\mathit{Propose}(k) := (up, j)$
$S := k + 1$
$\forall k < S$ such that $\mathit{Propose}(k) = \texttt{nil}$ do
$\mathit{Propose}(k) := \texttt{dummy}$

output Report-update$_i$
Pre: $\mathit{Decision}(k) = (up, i)$
$\mathit{Propose}(k) = (up, i)$
$X = (O, k - 1)$
Eff: $X := (up(O), k)$

internal Update$_i$
Pre: $\mathit{Decision}(k) = (up, j)$
$j \neq i$
$X = (O, k - 1)$
Eff: $X := (up(O), k)$

internal RePropose$(k)_i$
Pre: $\mathit{Propose}(k) = (up, i)$
$\mathit{Decision}(k) \neq (up, i)$
$\mathit{Decision}(k) \neq \texttt{nil}$
$\mathit{Reproposed}(k) = \texttt{false}$
Eff: $\mathit{Reproposed}(k) := \texttt{true}$
$\mathit{Propose}(S) := (up, i)$
$S := S + 1$

input Decide$(k, (up, j))_i$
Eff: $\mathit{Decision}(k) := (up, j)$

Figure 8-2: Automaton DATAREPLICATION for process i (part 2)

A read request is satisfied by simply returning the status of the local copy of the object. Action Request-read$_i$ sets the variable *CurRead* to the current status O of the local copy and action Report-read$(O)_i$ reports this status to the client.

To satisfy an update request the requested operation must be agreed upon by all processes. Hence it has to be proposed in instances of MULTIPAXOS until it is the outcome of an instance. A Request-update$(up)_i$ action has the effect of setting $Propose(k)$, where $k = S$, to (up, i); action Init$(k, (up, j))_i$[1] is then executed so that process i has (up, j) as initial value in the k-th instance of PAXOS. However since process i may be not the leader it has to broadcast a message to request the leader to run the k-th instance (the leader may be waiting for an initial value for the k-th instance). Action SendWantPaxos$_i$ takes care of this by broadcasting a "WantPaxos" message specifying the instance k and also the proposed operation (up, i) so that any process that receives this message (and thus also the leader) and has its $Propose(k)$ value still undefined will set it to (up, i). Action RecWantPaxos takes care of the receipt of "WantPaxos" messages. Notice that whenever the receipt of a "WantPaxos" message results in setting $Propose(k)$ to the operation specified in the message, possible gaps in the sequence of proposed operation are filled with a *dummy* operation which has absolutely no effect on the object O. This avoids gap in the sequence of update operations.

When the k-th instance of PAXOS reaches consensus on a particular update operation (up, i), the update can be applied to the object, given that the $(k-1)$-*th* update operation has been applied to the object, and the result of the update can be given back to the client that requested the update operation. This is done by action Report-update$(r)_i$. Action Update$_i$ only updates the local copy without reporting anything to the client if the operation was not requested by client i. If process i proposed an operation *up* as the k-th one and another operation is installed as the k-th one, then process i has to re-propose operation *up* in another instance of PAXOS. This

[1] Notice that we used the identifier j since process i may propose as its initial value the operation of another process j if it knows that process j is proposing that operation (see actions SendWantPaxos$_i$ and RecWantPaxos$_i$).

is done in action RePropose$_i$. Notice that process i has to re-propose only operations that it proposed, i.e., operations of the form (up, i).

State variables. *Propose* is an array used to store the operations to propose as initial values in the instances of PAXOS. *Decision* is an array used to store the outcomes of the instances of PAXOS. The integer S is the index of the first undefined entry of the array *Propose*. This array is kept in such a way that it is always defined up to $Propose(S-1)$ and is undefined from $Propose(S)$. Variable X describes the current state of the object. Initially the object is in its initial state O_0. The DATAREPLICATION$_i$ automaton keeps an updated copy of the object, together with the index of the last operation applied to the object. Initially the object is described by $(O_0, 0)$. Let O_k be the state of the object after the application to O_0 of the first k operations. When variable $X = (O, k)$, we have that $O = O_k$. When the outcome $Decision(k+1) = (up, i)$ of the $(k+1)$-*th* instance of Paxos is known and current state of the object is (O, k), the operation up can be applied and process i can give back a response to the client that requested the operation.

Variable *CurRead* is used to give back the report of a read. Variable $Proposed(k)$ is a flag indicating whether or not an Init$(k, v)_i$ action for the k-th instance has been executed, so that the Init$(k, v)_i$ action is executed at most once (though executing this action multiple times does not affect PAXOS). Similarly $Reproposed(k)$ is a flag used to re-propose only once an operation that has not been installed. Notice that an operation must be re-proposed only once because a re-proposed action will be re-proposed again if it is not installed.

8.3.2 Correctness and analysis

We do not prove formally the correctness of the DATAREPLICATION algorithm. By correctness we mean that sequential consistency is never violated. Intuitively, the correctness of DATAREPLICATION follows from the correctness of MULTIPAXOS. Indeed all processes agree on each update operation to apply to the object: the outcomes of the various instances of PAXOS give the sequence of operations to apply to the object and each process has the same sequence of update operations.

Theorem 8.3.1 *Let α be an execution of the system consisting of* DATAREPLICATION *and* MULTIPAXOS. *Let β be the subsequence of α consisting of the request/report events and assume that β is complete. Then β is sequentially consistent.*

Proof sketch: To see that β is sequentially consistent it is sufficient to give an atomic request/report sequence γ such that $\gamma|i = \beta|i$, for each client i. The sequence γ can be easily constructed in the following way: let $up_1, up_2, up_3, ...$ be the sequence of update operations agreed upon by all the processes; let γ' be the request/report sequence Request-update$(up_1)_{i_1}$,Report-update$_{i_1}$, Request-update$(up_2)_{i_2}$, Report-update$_{i_2}$, ...; then γ is the sequence obtained by γ' by adding Request-read, Report-read events in the appropriate places (i.e., if client i requested a read when the status of the local copy was O_k, then place Request-read$_i$, Report-read$(O_k)_i$, between Report-update$_{i_k}$ and Request-update$(up_{k+1})_{i_{k+1}}$). ■

Liveness is not guaranteed. Indeed it is possible that an operation is never satisfied because new operations could be requested and satisfied. Indeed PAXOS guarantees validity but any initial value can be the final output value, thus when an operation is re-proposed in subsequent instances, it is not guaranteed that eventually it will be the outcome of an instance of PAXOS if new operations are requested. A simple scenario is the following. Process 1 and process 2 receive requests for update operations up_1 and up_2, respectively. Instance 1 of PAXOS is run and operation up_2 proposed by process 2 is installed. Thus process 1 re-proposes its operation in instance 2. Process 3 has, meanwhile, received a request for update operation up_3 and proposes it in instance 2. The operation up_3 of process 3 is installed in instance 2. Again process 1 has to re-propose its operation in a new instance. Nothing guarantees that process 1 will eventually install its operation up_1 if other processes keep proposing new operations. This problem could be avoided by using some form of priority for the operations to be proposed by the leader in new instances of PAXOS.

The algorithm exhibits the same fault tolerance and liveness properties of PAXOS: process stop and recovery, message loss, duplication and reordering and timing failures. However, as in PAXOS, to get progress it is necessary that the system executes a long enough nice execution fragment.

8.4 Concluding remarks

The application of MULTIPAXOS to data replication that we have presented in this chapter is intended only to show how MULTIPAXOS can be used to implement a data replication algorithm. A better data replication algorithm based on MULTIPAXOS can certainly be designed. We have not provided a proof of correctness of this algorithm; also the performance analysis is not given. There is work to be done to obtain a good data replication algorithm.

For example, it should be possible to achieve liveness by using some form of priority for the operations proposed in the various instances of PAXOS. The easiest approach would use a strategy such that an operation that has been re-proposed more than another one, has priority, that is, if the leader can choose among several operations, it chooses the one that has been re-proposed most. This should guarantee that requested operations do not “starve” and are eventually satisfied.

In this chapter we have only sketched how to use PAXOS to implement a data replication algorithm. We leave the development of a data replication algorithm based on PAXOS as future work.

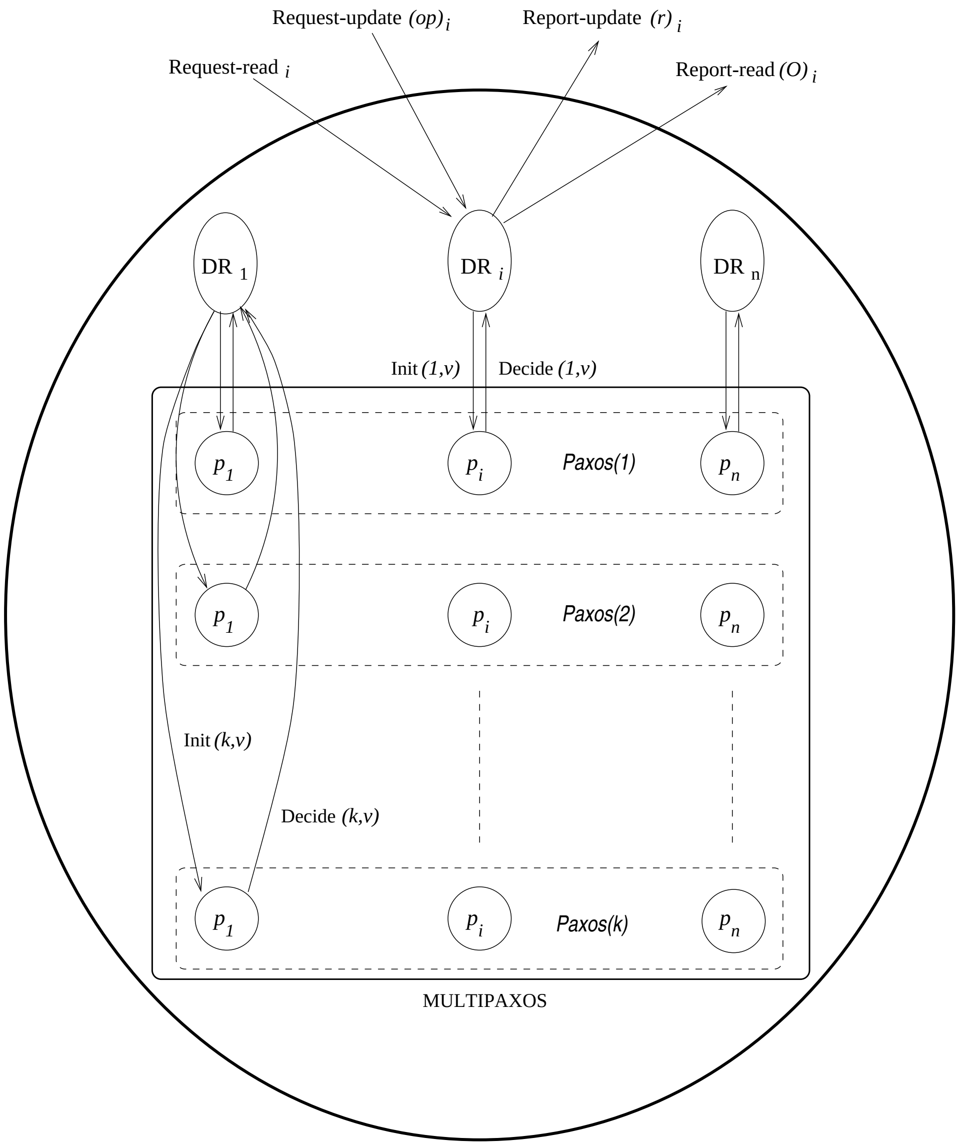

Figure 8-3: Data replication

Chapter 9

Conclusions

The consensus problem is a fundamental problem in distributed systems. It plays a key role in practical problems involving distributed transactions. In practice the components of a distributed systems are subject to failures and recoveries, thus any practical algorithm should cope as much as possible with failures and recoveries. PAXOS is a highly fault-tolerant algorithm for reaching consensus in a partially synchronous distributed system. MULTIPAXOS is a variation of PAXOS useful when consensus has to be reached on a sequence of values. Both PAXOS and MULTIPAXOS were devised by Lamport [29].

The PAXOS algorithm combines high fault-tolerance with efficiency; safety is maintained despite process halting and recovery, messages loss, duplication and reordering, and timing failures; also, when there are no failures nor recoveries and a majority of processes are alive for a sufficiently long time, PAXOS reaches consensus using linear, in the number of processes, time and messages.

PAXOS uses the concept of a leader, i.e., a distinguished process that leads the computation. Unlike other algorithms whose correctness is jeopardized if there is not a unique leader, PAXOS is safe also when there are no leaders or more than one leader; however to get progress there must be a unique leader. This nice property allows us to use a sloppy leader elector algorithm that guarantees the existence of a unique leader only when no failures nor process recoveries happen. This is really important in practice, since in the presence of failures it is not practically possible to provide a reliable leader elector (this is due to the difficulty of detecting failures).

Consensus algorithms currently used in practice are based on the 2-phase commit algorithm (e.g., [2, 25, 41, 48], see also [22]) and sometime on the 3-phase commit algorithm (e.g. [47, 48]). The 2-phase commit protocol is not at all fault tolerant. The reason why it is used in practice is that it is very easy to implement and the probability that failures affect the protocols is low. Indeed the time that elapses from the beginning of the protocol to its end is usually so short that the possibility of failures becomes irrelevant; in small networks, messages are delivered almost instantaneously so that a 2-phase commit takes a very short time to complete; however the protocol blocks if failures do happen and recovery schemes need to be invoked. Protocols that are efficient when no failures happen yet highly fault tolerant are necessary when the possibility of failures grows significantly, as happens, for example, in distributed systems that span wide areas. The PAXOS algorithm satisfy both requirements.

We believe that PAXOS is the most practical solution to the consensus problem currently available. To support the previous statement it would be nice to have an implementation of PAXOS.

In the original paper [29], the PAXOS algorithm is described as the result of discoveries of archaeological studies of an ancient Greek civilization. That paper contains a sketch of a proof of correctness and a discussion of the performance analysis. The style used for the description of the algorithm often diverts the reader's attention. Because of this, we found the paper hard to understand and we suspect that others did as well. Indeed the PAXOS algorithm, even though it appears to be a practical and elegant algorithm, seems not widely known or understood, either by distributed systems researchers or distributed computing theory researchers.

In this thesis we have provided a new presentation of the PAXOS algorithm, in terms of I/O automata; we have also provided a correctness proof and a time performance and fault-tolerance analysis. The correctness proof uses automaton composition and invariant assertion methods. The time performance and fault-tolerance analysis is conditional on the stabilization of the system behavior starting from some point in an execution. Stabilization means that no failures nor recoveries happen after the stabilization point and a majority of processes are alive for a sufficiently long time.

We have also introduced a particular type of automaton model called the Clock GTA. The Clock GTA model is a particular type of the general timed automaton (GTA) model. The GTA model has formal mechanisms to represent the passage of time. The Clock GTA enhances those mechanisms to represent timing failures. We used the Clock GTA to provide a technique for practical time performance analysis based on the stabilization of the physical system. We have used this technique to analyze PAXOS.

We also have described MULTIPAXOS and discussed an example of how to use MULTIPAXOS for data replication management. Another immediate application of PAXOS is to distributed commit. PAXOS bears some similarities with the 3-phase commit protocol; however 3-phase commit, needs a failure detector stronger than the one needed by PAXOS.

Our presentation of PAXOS has targeted the clarity of presentation of the algorithm; a practical implementation does not need to be as modular as the one we have presented. For example, we have separated the leader behavior of a process into two parts, one that takes care of leading a round and another one that takes care of broadcasting a reached decision; this has resulted in the duplication of state information and actions. In a practical implementation it is not necessary to have such a separation. Also, a practical algorithm could use optimizations such as message retransmission, or waiting larger time-out intervals before abandoning a round, so that a message loss or a little delay does not force the algorithm to start a new round.

Further directions of research concern improvements of PAXOS. For example it is not clear whether a clever strategy for electing the leader can help in improving the overall performance of the algorithm. We used a simple leader election strategy which is easy to implement, but we do not know if more clever leader election strategies may positively affect the efficiency of PAXOS. Also, it would be interesting to provide performance analysis for the case when there are failures, in order to measure how badly the algorithm can perform. For this point, however, one should keep in mind that PAXOS does not guarantees termination in the presence of failures. We remark that allowing timing failures and process stopping failures the problem is unsolvable. However in some cases termination is achieved even in the presence of failures, e.g., only a few messages are lost or a few processes stop.

It would be interesting to compare the use of PAXOS for data replication with other related algorithms such as the data replication algorithm of Liskov and Oki. Their work seems to incorporate ideas similar to the ones used in PAXOS.

Also the virtual synchrony group communication scheme of Fekete, Lynch and Shvartsman [16] based on previous work by Amir *et. al.* [3], Keidar and Dolev [24] and Cristian and Schmuck [7], uses ideas somewhat similar to those used by PAXOS: quorums and timestamps (timestamps in PAXOS are basically the round numbers).

Certainly a further step is a practical implementation of the PAXOS algorithm. We have shown that PAXOS is very efficient and fault tolerant in theory. While we are sure that PAXOS exhibits good performance from a theoretical point of view, we still need the support of a practical implementation and the comparison of the performance of such an implementation with existing consensus algorithms to affirm that PAXOS is the best currently available solution to the consensus problem in distributed systems.

We recently learned that Lee and Thekkath [33] used PAXOS to replicate state information within their Petal system which implements a distributed file server. In the Petal system several servers each with several disks cooperate to provide to the users a virtual, big and reliable storage unit. Virtual disks can be created and deleted. Servers and physical disks may be added or removed. The information stored on the physical disks is duplicated to some extent to cope with server and or disk crashes and load balancing is used to speed up the performance. Each server of the Petal system needs to have a consistent global view of the current system configuration; this important state information is replicated over all servers using the PAXOS algorithm.

Appendix A

Notation

This appendix contains a list of symbols used in the thesis. Each symbol is listed with a brief description and a reference to the pages where it is defined.

n	number of processes in the distributed system. (37)
$\mathcal{I}$	ordered set of n process identifiers. (37)
ℓ	time bound on the execution of an enabled action. (37)
d	time bound on the delivery of a message. (37)
V	set of initial values. (47)
$\mathcal{R}$	set of round numbers. A round number is a a pair (x, i),
	where $x \in \mathcal{I}$ and $x \in \mathbb{N}$. Round numbers are totally ordered. (65)
$\texttt{Hleader}(r)$	history variable. The leader of round r. (80)
$\texttt{Hvalue}(r)$	history variable. The value of round r. (80)
$\texttt{Hfrom}(r)$	history variable. The round from which the value of round r is taken. (80)
$\texttt{Hinfquo}(r)$	history variable. The info-quorum of round r. (80)
$\texttt{Haccquo}(r)$	history variable. The accepting-quorum of round r. (80)
$\texttt{Hreject}(r)$	history variable. Processes committed to reject round r. (80)
$\mathcal{R}_S$	set of round numbers of rounds for which `Hleader` is set. (82)
$\mathcal{R}_V$	set of round numbers of rounds for which `Hvalue` is set. (82)
t^i_α	time of occurrence of $\mathrm{Init}(v)_i$ in α. (93)
T^i_α	max of $4\ell + 2n\ell + 2d$ and $t^i_\alpha + 2\ell$. (82)

S_{CHA}	distributed system consisting of $\text{CHANNEL}_{i,j}$, for $i, j \in \mathcal{I}$ (42)
S_{DET}	distributed system consisting of S_{CHA} and DETECTOR_i, for $i \in \mathcal{I}$ (52)
S_{LEA}	distributed system consisting of S_{DET} and LEADERELECTOR_i, for $i \in \mathcal{I}$. (56)
S_{BPX}	distributed system consisting of S_{CHA} and BPLEADER_i, BPAGENT_i and BPSUCCESS_i for $i \in \mathcal{I}$. (80)
S_{PAX}	distributed system consisting of S_{LEA} and BPLEADER_i, BPAGENT_i, BPSUCCESS_i and STARTERALG_i for $i \in \mathcal{I}$. (100)

regular time-passage step, a time-passage step $\nu(t)$ that increases the local clock of each Clock GTA by t. (26)

regular execution fragment, an execution fragment whose time-passage steps are all regular. (26)

stable execution fragment, a regular execution fragment with no process crash or recoveries and no loss of messages. (38–42)

nice execution fragment, a stable execution fragment with a majority of processes alive. (43)

start of a round, is the execution of action NewRound for that round. (91)

end of a round, is the execution of action RndSuccess for that round. (91)

successful round, a round is successful when it ends, i.e., when action RndSuccess for that round is executed. (91)

Bibliography

[1] P. Alsberg, J. Day, A principle for resilient sharing of distributed resources. In Proc. of the 2^{nd} *International Conference on Software Engineering*, pp. 627–644, Oct. 1976.

[2] A. Adya, R. Gruber, B. Liskov and U. Maheshwari, Efficient Optimistic Concurrency Control using Loosely Synchronized Clocks, SIGMOD, San Jose, CA, pp. 23–34. May 1995.

[3] Y. Amir, D. Dolev, P. Melliar-Smith and L. Moser, Robust and Efficient Replication Using Group Communication, Technical Report 94-20, Department of Computer Science, Hebrew University, 1994.

[4] T.D. Chandra, V. Hadzilacos, S. Toueg, The weakest failure detector for solving consensus, in *Proceedings of the* 11^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pp. 147–158, Vancouver, British Columbia, Canada, August 1992.

[5] T.D. Chandra, S. Toueg, Unreliable failure detector for asynchronous distributed systems, Proceedings of PODC 91, pp. 325–340. in *Proceedings of the* 10^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pp. 325–340, August 1991.

[6] E.C. Cooper, Replicated distributed programs. UCB/CSD 85/231, University of California, Berkeley, CA, May 1985.

[7] F. Cristian and F. Schmuck, Agreeing on Processor Group Membership in Asynchronous Distributed Systems, Technical Report CSE95-428, Department of Computer Science, University of California San Diego.

[8] D. Dolev, Unanimity in an unknown and unreliable environment. *Proc. 22nd IEEE Symposium on Foundations of Computer Science*, pp. 159–168, 1981

[9] D. Dolev, The Byzantine generals strike again. *J. of Algorithms* vol. 3 (1), pp. 14–30, 1982.

[10] D. Dolev, C. Dwork, L. Stockmeyer, On the minimal synchrony needed for distributed consensus, *J. of the ACM*, vol. 34 (1), pp. 77-97, January 1987.

[11] C. Dwork, N. Lynch, L. Stockmeyer, Consensus in the presence of partial synchrony, *J. of the ACM*, vol. 35 (2), pp. 288–323, April 1988.

[12] D.L. Eager and K.C. Sevcik, Achieving robustness in distributed database systems, *ACM Trans. on Database Systems* vol. 8 (3), pp. 354–381, September 1983.

[13] A. El Abbadi, D. Skeen, F. Cristian, An efficient fault-tolerant protocol for replicated data management, Proc. of the 4^{th} ACM SIGACT/SIGMOD Conference on Principles of Database Systems, 1985.

[14] A. El Abbadi, S. Toueg, Maintaining availability in partitioned replicated databases, Proc. of the 5^{th} ACM SIGACT/SIGMOD Conference on Principles of Data Base Systems, 1986

[15] A.Fekete, F. Kaashoek, N. Lynch, Implementing Sequentially-Consistent Shared Objects Using Group and Point-to-Point Communication, n the 15th International Conference on Distributed Computing Systems (ICDCS95), pp. 439-449, Vancouver, Canada, May/June 1995, IEEE. Abstract/Paper. Also, Technical Report MIT/LCS/TR-518, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, June 1995.

[16] A. Fekete, N. Lynch, A. Shvartsman, Specifying and using a partitionable group communication service, manuscript, 1997.

[17] M.J. Fischer, The consensus problem in unreliable distributed systems (a brief survey). Rep. YALEU/DSC/RR-273. Dept. of Computer Science, Yale Univ., New Have, Conn., June 1983.

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

[19] Z. Galil, A. Mayer, M. Yung, Resolving the message complexity of Byzantine agreement and beyond, Proceedings of the 37^{th} *IEEE Symposium on Foundations of Computer Science (FOCS)*, pp. 724–733, 1995.

[20] D.K. Gifford, Weighted voting for replicated data. Proc. of the $7th$ ACM Symposium on Operating Systems Principles, *SIGOPS Operating Systems Review*, vol. 13 (5), pp. 150–162. December 1979.

[21] J. N. Gray. Notes on data base operating systems. In R. Bayer, R. M. Graham, and G, Seegmuller, editors, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, pp. 450–463, Philadelphia, Pennsylvania, June 1978.

[22] J. N. Gray, A. Reuter. Transaction Processing: Concepts and Techniques, Morgan Kaufmann Publishers, Inc., San Mateo CA, 1993.

[23] M.P. Herlihy, A quorum-consensus replication method for abstract data types. *ACM Trans. on Computer Systems* vol. 4 (1), 32–53, February 1986.

[24] I. Keidar and D. Dolev, Efficient Message Ordering in Dynamic Networks, in *Proc. of 15th Annual ACM Symp. on Princ. of Distr. Comput.*, pp. 68-76, 1996.

[25] A. Kirmse, Implementation of the two-phase commit protocol in Thor, S.M. Thesis, Lab. for Computer Science, Massachusetts Institute of Technology, May 1995.

[26] L. Lamport, Proving the correctness of multiprocess programs. *IEEE Transactions on Software Eng.*, Vol. SE-3, pp. 125–143, Sept. 1977.

[27] L. Lamport, On interprocess communication, Part I: Basic formalism. *Distributed Computing*, 1(2), pp. 77–85, Apr. 1986.

[28] L. Lamport, On interprocess communication, Part II: Algorithms. *Distributed Computing*, 1(2), pp. 86–101, Apr. 1986.

[29] L. Lamport, The part-time parliament. *ACM Transactions on Computer Systems*, 16 (2), pp. 133–169, May 1998. Also Research Report 49, Digital Equipment Corporation Systems Research Center, Palo Alto, CA, September 1989.

[30] L. Lamport, R. Shostak, M. Pease, The Byzantine generals problem, *ACM Trans. on Program. Lang. Syst.* 4 (3), 382–401, July 1982.

[31] B. Lampson, W. Weihl, U. Maheshwari, Principle of Computer Systems: Lecture Notes for 6.826, Fall 1992, Research Seminar Series MIT/LCS/RSS 22, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, July 1993.

[32] B. Lampson, How to build a highly available system using consensus, in *Proceedings of the tenth international Workshop on Distributed Algorithms* WDAG 96, Bologna, Italy, pp. 1–15, 1996.

[33] E.K. Lee, C.A. Thekkath, Petal: Distributed virtual disks, In Proceedings of the *Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 84-92, Cambridge, MA, October 1996.

[34] B. Liskov, B. Oki, Viewstamped replication: A new primary copy method to support highly-available distributed systems, in *Proceedings of the* 7^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pp. 8–17, August 1988.

[35] N. Lynch, Distributed Algorithms, Morgan Kaufmann Publishers, San Francisco, 1996.

[36] N. Lynch, H. Attiya. Using mappings to prove timing properties. *Distributed Computing*, 6(2):121–139, September 1992.

[37] N. Lynch, M.R. Tuttle, An introduction to I/O automata, *CWI-Quarterly*, 2 (3), 219–246, CWI, Amsterdam, The Netherlands, September 89. Also Technical Memo MIT/LCS/TM-373, Lab. for Computer Science, MIT, Cambridge, MA, USA, Nov 88.

[38] N. Lynch, F. Vaandrager. Forward and backward simulations for timing-based systems. in *Real-Time: Theory in Practice*, Vol. 600 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 397–446, 1992.

[39] N. Lynch, F. Vaandrager. Forward and backward simulations—Part II: Timing-based systems. Technical Memo MIT/LCS/TM-487.b, Lab. for Computer Science, MIT, Cambridge, MA, USA, April 1993.

[40] N. Lynch, F. Vaandrager. Actions transducers and timed automata. Technical Memo MIT/LCS/TM-480.b, Lab. for Computer Science, MIT, Cambridge, MA, USA, Oct 1994.

[41] C. Mohan, B. Lindsay, R. Obermarck, Transaction Management in R* Distributed Database Management System, *ACM Transactions on Computer Systems*, Vol. 11, No. 4, pp. 378–396, December 1986.

[42] M. Merritt, F. Modugno, and M.R. Tuttle. Time constrained automata. *CONCUR 91: 2nd International Conference on Concurrency Theory*, Vol. 527 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 408–423, 1991.

[43] B. Patt, A theory of clock synchronization, Ph.D. Thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139, Oct 1994.

[44] B. Patt-Shamir, S. Rajsbaum, A theory of clock synchronization, in *Proceedings of the* 26^{th} *Symposium on Theory of Computing*, May 1994.

[45] M. Pease, R. Shostak, L. Lamport, Reaching agreement in the presence of faults, *Journal of the ACM* 27 (2), 228–234, April 1980.

[46] D. Skeen, D.D. Wright, Increasing availability in partitioned database systems, TR 83-581, Dept. of Computer Science, Cornell University, Mar 1984.

[47] D. Skeen, Nonblocking Commit Protocols. Proceedings of the ACM SIGMOD International Conference on Management of Data, pp. 133–142, May 1981.

[48] A. Z. Spector, Distributed transaction processing and the Camelot system, CMU-CS-87-100, Carnegie-Mellon Univ. Computer Science Dept., 1987.

[49] G. Varghese, N. Lynch, A tradeoff between safety and liveness for randomized coordinated attack protocols, *Information and Computation*, vol 128, n. 1 (1996), pp. 57–71. Also in *Proceedings of the* 11^{th} *Annual ACM Symposium on Principles of Distributed Computing*, pp. 241–250, Vancouver, British Columbia, Canada, August 1992.