A Formal Approach to Fault Tolerant Distributed Consensus

Rachele Fuzzati

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

LABORATOIRE DES MÉTHODES DE PROGRAMMATION
INSTITUT D'INFORMATIQUE FONDAMENTALE
FACULTÉ INFORMATIQUE ET COMMUNICATION
ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

Contents

List of Figures

List of Tables

Riassunto

L'espressione *Consenso distribuito* denota il problema far raggiungere un accordo sulla scelta di un valore ad un gruppo di processi distribuiti. Tali processi comunicano scambiandosi messaggi attraverso un qualche mezzo di comunicazione e possono essere anche situati a grande distanza gli uni dagli altri (da cui il termine distribuiti). È stato provato che tale problema è impossibile da risolvere in sistemi asincroni quando anche un solo processo può subire un guasto, i.e., smettere di funzionare. Dato che il problema di raggiungere il Consenso tra processi è molto frequente nel campo del calcolo distribuito, sono stati proposti molti algoritmi per risolverlo eludendo il risultato di impossibilità attraverso l'introduzione nel sistema di alcuni elementi di sincronia. Tali algoritmi sono tradizionalmente espressi in linguaggio naturale o in pseudo-codice, cosa che a volte genera ambiguità riguardo al loro contenuto e alle loro prove di correttezza.

In questa tesi proponiamo una semplice, ma efficace metodologia per fornire descrizioni e prove di correttezza formali di algoritmi di Consenso distribuiti. Tale metodologia è basata sull'utilizzo di regole di inferenza, necessita di semplici basi teoriche per essere compresa e segue da vicino il modo in cui gli algoritmi sono espressi in pseudo-codice, risultando quindi intuitiva per gli utilizzatori. Per mostrare la veridicità di quanto affermato, ci serviamo della nostra metodologia per formalizzare due dei maggiori algoritmi di Consenso distribuito: l'algoritmo di Chandra e Toueg e l'algoritmo denominato Paxos.

In seguito, utilizzando la nostra descrizione rigorosa, proviamo che tali algoritmi garantiscono il rispetto delle proprietà di Validità, Accordo e Terminazione che ogni soluzione al problema del Consenso dovrebbe garantire. In effetti, questo esercizio di dimostrazione rivela dei risultati interessanti. Notiamo infatti che l'algoritmo di Chandra-Toueg e Paxos si assomigliano sotto molteplici aspetti e che le loro prove di correttezza possono essere svolte in modo quasi identico. Tuttavia, mentre possiamo dimostrare che l'algoritmo di Chandra-Toueg è corretto dal punto di vista delle tre proprietà, scopriamo che Paxos non dà garanzie di terminazione. Questa scoperta genera una domanda filosofica: possiamo considerare Paxos un algoritmo di Consensus oppure no?

Parole chiave: metodi formali, algoritmi distribuiti, consenso distribuito, sistemi di transizioni, prove di correttezza, terminazione.

Abstract

The term *distributed Consensus* denotes the problem of getting a certain number of processes, that could be far away from each other and that exchange messages through some communication means, to all agree on the same value. This problem has been proved impossible to solve in asynchronous settings when at least one process can crash, i.e., stop working. Since the problem of reaching Consensus among processes is recurrent in the domain of distributed computation, many algorithms have been proposed for solving it, circumventing the impossibility result through the introduction of some kind of synchrony in the system. Such algorithms are traditionally expressed in natural language or in pseudocode, thus sometimes generating ambiguities on their contents and on their correctness proofs.

In this thesis, we propose a simple, yet efficient way of providing formal descriptions and proofs of distributed Consensus algorithms. Such method is based on the use of inference rules, it requires very little prior knowledge in order to be understood, and follows closely the way algorithms are expressed in pseudocode, thus being intuitive for the users. To show the validity of our claims, we use our method to formalize two of the major distributed Consensus algorithms, namely the Chandra-Toueg and the Paxos algorithms.

Using our rigorous description, we then formally prove that such algorithms guarantee the respect of the Validity, Agreement and Termination properties that every solution to the Consensus problem should provide. This proving exercise actually reveals interesting results. We see that the Chandra-Toueg and the Paxos algorithms have strong points of resemblance and their correctness proofs can be carried out in very similar manners. However, while the Chandra-Toueg algorithm proves to be correct under the point of view of the three properties, we discover that Paxos does not give any guarantee of terminating. This generates a philosophical question: should such algorithm be considered a Consensus algorithm or not?

Keywords: formal methods, distributed algorithms, distributed consensus, transition systems, correctness proofs, termination.

Acknowledgements

It is always difficult for me to write an acknowledgment section because I always fear to transmit an order of importance when listing people...

I would like to thank my advisers: Prof. Uwe Nestmann for accepting me in his group and for his support in ”dissecting” distributed Consensus, and Prof. André Schiper for accepting to be my co-supervisor when Uwe left EPFL.

A special mention goes to Prof. Massimo Merro who started the Consensus adventure with us and put a lot of effort into making us eventually reach termination for our journal paper.

I would like to thank all my course-mates of the EPFL Pre-Doctoral School with whom I shared not only the troubles of getting started in a new environment, but also many happy moments. In particular, I would like to mention Shai, who has been very close to me any time I needed him.

The group of ”vrais potes” (Philippe Altherr, Sébastien Briais, Vincent Cremet, Gilles Dubochet and Michel Schinz) was both a great support for solving troubles during working hours and a good companion for relaxing outside EPFL. In particular, Vincent spent many hours patiently listening to my stories about Consensus and giving me his useful insights. He dedicated two weeks during his summer holidays to the construction of a simulator based on our inference rule description of the Chandra-Toueg algorithm and provided very helpful comments on my final report.

I would also like to thank all the other (past and present) members of LAMP: Prof. Martin Odersky, Johannes Borgström, Daniel Bünzli, Iulian Dragos, Burak Emir, Philipp Haller, Simon Kramer, Sebastian Maneth, Sean McDirmid, Stéphane Micheloud, Nikolay Mihaylov, Christine Röckl, Lex Spoon, Erik Stenman and Matthias Zenger for making our lab one of the most envied ones in EPFL. In particular, during his staying in Lausanne, Johannes has been a great companion of dissertation on anything, from I/O Automata to the meaning of life.

I would like to thank all the members of the Funny Dog Bussigny

Agility Club for helping me integrating in Switzerland, for giving me a good reason to learn French and for sharing with me many carefree days. In particular, I would like to thank Nadia, who has been extremely sensible, ready and kind any time I needed help, and Louis, who has been my ever tired companion of endless trips and sleepless nights before competitions, making me remember that there is life outside research.

Last, but not least, I would like to thank all my family for the constant love and support they have always given me.

Chapter 1

Introduction

Our world is more and more dominated by technology. Computers, mobile phones, communication networks have with no doubt become man's best helpers. It is quite common to confide them important or delicate information, trusting that we will be able to retrieve and modify our data at any time. Moreover, with the advent of globalization, it is more and more frequent to have companies showing their main siege in Europe, the Research & Development group in the USA and the production site in Asia, all confidently sharing and manipulating some common information.

However, well hidden behind the availability and reliability that appear to the final users, there are a great number of technologies and engineering solutions built to mask the problems related to information storage, data synchronization, transmission of information and so on. Let us take the example of storage. We all agree on the fact that in this world there is no infallible electronic equipment. Thus, if we store our data only on one machine, in case we run into technical problems we are very likely to lose all our information. In this case, the freedom from faults is achieved through the use of replication. The more the stored data is important, the more replicas containing the data are usually created. In this scenario, as well as in the case of globalized companies, if a user wants to make a simple change in the stored data, many repositories are in reality affected by the operation and many messages have to be exchanged between the repositories in order to ensure information consistency. This necessary communication introduces additional elements of failure: the possible delay and loss of messages.

For this kind of problem, and for any other situation in which some system element needs to take a decision in concert with other elements, there exist solutions that are called Consensus algorithms. Every Consensus algorithm is able to mask the failures happening in the system by having all elements agreeing on the same answer. Thus, no matter which element we ask, we always get the same answer.

To break the ice and introduce our research topic, we present here our favorite example when explaining our work to everyday people.

1.1 A Toy Example

Imagine you and your friends want to arrange a meeting together and you want to do so only by using a Short Message System [1]. SMSs are a cheap way of communicating, but they give no guarantee about reception. In fact, the telephone company can delay the delivery of messages, or sometimes even lose them. Moreover, some of your friends might be busy for some time during the day and not notice immediately that an SMS has come to their mobile. Or it can even happen that one of your friends suddenly receives an emergency call, shut down his mobile phone and lose interest in the experiment. The question is: is it possible to find a time and a place such that everyone of your friends that has not shut down the phone knows about? This question is not a facetiousness, since it is exactly what happens in millions of computers every day. The problem is called Consensus and it has been proved to have no solution in the settings we just described. In fact, since you are not allowed to make calls, you have no way of distinguishing the friends that are temporarily busy from those who have quit the experiment. Taking the risk of deciding for a date without consulting each friend who is still running the experiment might lead to some people not being able to express their preferences and therefore not being able to come to the appointment.

Thus, in order to solve your Consensus problem, you need to introduce a bit of synchrony in the system (in your case an acknowledgment of message reception is enough) that allows you to distinguish between friends in and out of the experiment. Then you need to define a procedure (an *algorithm*) that you and your friends are going to follow every time you want to set up a meeting in this way. This procedure must be carefully designed to ensure that the venue is chosen among the venues initially proposed by your friends, that all the friends that have not quit the experiment agree on the same date and location, and finally . . . that the meeting is eventually going to be set.

Of course, the procedure needs to be written down as a set of instructions to be carefully followed and each one of your friends should get a

[1]From now on, we will indicate with SMS any message sent through such system.

copy of it. Since your friends share with you a lot of interests, but they do not necessarily share the same mother tongue or the same way of seeing things, writing sentences like ”Repeat the following as many times as you want: send the message to any participant.” or ”You have the right to suspect any of the participants to have quit the experiment.” might rise questions like ”Am I allowed to send all the time the message to Giovanni and never send it to the others? Isn’t it a bit useless?” or ”Can I suspect also myself? If I’m the one who’s taking an action, how can I have quit the experiment?” Thus, you must be quite precise in your description of the procedure, and you should try to be as rigorous as possible. However, if you use well designed but complex methods for your description, you run the risk of having some of your friends complaining on their lack of time to learn and understand the method. Thus, you will end up with a group of participants not being able to understand your description. Your other choice is to use the basic mathematical knowledge of your friends to write down a description that results a bit ad hoc to your algorithm, but that allows everybody to quickly grasp the procedure without leaving any ambiguity. Such description can then be used by your most curious friends as a starting point for expressing the procedure using the more complex methods your other friends did not have time to study.

Since in every group there is always at least a slightly paranoid component, one of your friends might ask you whether you are sure that the procedure you are proposing actually guarantees to give the expected results, i.e., whether you are confident that you will eventually be able to decide and meet all at the same place at the same time. If you had simply expressed the procedure with words, without formalizing it, at this point you could simply give intuitive reasons on why your friend should not worry about finding himself alone with Francesco in one place while all the others have fun somewhere else. Since you have given a formalization of your procedure, you can prove that the desired properties are respected by showing it with mathematical and rigorous tools. This kind of proof will result much more convincing for your friend (and for you!) than any other informal argument.

But what happens if, while trying to prove the correctness of the procedure you propose, you realize that there are some cases in which there is no guarantee of eventually reaching a decision? Could you still claim that the procedure solves the desired problem or not? Since you can prove that your proposal is safe (i.e., the chosen venue will always be one that was initially proposed by some friend and there will never be two subgroups of people meeting in two different places), you can be sure that if you agree on some date, then such date will be one that presents the required specifications. On the other hand, if you wanted to organize the meeting to discuss of something important, it might be quite annoying if you can never eventually manage to decide on its venue.

1.2 Getting more Serious

Distributed computing is about multiple distributed processes interacting with each other in order to reach a common objective. Very often, for economical or logistical reasons, processes communicate through non reliable means where the data exchanged can be delayed, lost or corrupted. Moreover, processes run on "real" machines that are by nature failure prone. Thus, the algorithms proposed to master the interactions between processes in distributed computing must be able to cope with communication and process failures.

We now understand the reasons why distributed algorithms look sometimes so twisted: it is the complexity of the system settings that forces to increase the complexity of the solution. Since distributed algorithms are often used in fundamental applications, we would like both their design and their implementation to be impeccable. But to succeed in this we must first of all find unambiguous ways of expressing the algorithm. In fact, only a description that does not leave place to questions and misunderstandings can allow a perfect correspondence between the algorithm description and its implementation. Another fundamental point is the possibility of giving proofs of correctness, i.e., showing that the algorithm provides exactly the properties it declares to guarantee. In fact, it is better to discover a problem at the time of designing the algorithm, rather than when there are already thousands of implementations running in the entire world. For this, the way in which the algorithm was originally described can help a great deal: if the description is non ambiguous, then it results easier to provide correctness proofs.

Clearly, each community has its preferred ways of representing things and it is a long-time tradition to write distributed algorithms using pure natural language or pseudo-code. However we believe that, given their value and intrinsic complexity, distributed algorithms deserve more detailed and formal descriptions than the ones currently in use. For this reason, we propose here a simple way of expressing distributed algorithms. We formally describe each component of the system: processes, communication means and other modules necessary for the execution. Then, recalling pseudo-code instructions, we describe each action of the system as a transition rule with very detailed premises and consequences. Each execution of the algorithm can be non ambiguously described through the subsequent application of the transition rules. The advantage of this method is that the formalization we propose is syntax free and does not refer to any form of programming language or logic. It is thus very intuitive and does not require any previous knowledge in order to be used and understood. Moreover, given the high level of details that is required to express the execution rules, our method forces the designer of the algorithm to deeply think about each action and its consequences and to describe them with all the particulars. This greatly helps eliminating interpretation or misunderstanding on the reader/implementer side. Last but not least, while pseudo-code provides us only with the description of what happens locally at each process site, our method forces us to take a global view of the system and of its evolution during the whole execution of the algorithm. Such global knowledge helps us having a clear idea of the process interactions and gives us more confidence when carrying out correctness proofs.

As we have seen in the toy example §1.1, our interest is for Consensus algorithms, particularly for those designed to run in asynchronous systems. Consensus algorithms guarantee that every execution originated from them satisfies the following properties [2]:

1. If a process decides a value v, then v was proposed by some process (*Validity*).
2. No two correct processes decide differently (*Agreement*).
3. Every correct process (eventually) decides some value (*Termination*).

Among the vast number of Consensus algorithms, we have chosen two representative ones. The first one, the Chandra and Toueg algorithm, is designed to cope with process crashes and message delays. The other one, the Part Time Parliament (better known as Paxos) algorithm, is made to cope with processes that can crash and subsequently recover, plus message delays and losses. At first sight, looking at their informal description and at the settings in which they are designed to work, the two algorithms look quite different from each other. However, once we formalize them with our method, we see that they are actually pretty similar. For this reason, in this report we have decided to present the two algorithms as two parallel rails. We have separated the various stages of description,

[2]Depending on the system setting, a correct process is a process that does not crash during the execution of the algorithm or that eventually recovers and does not any longer crash.

formalization and correctness proofs of the two algorithms in order to be able to put them one after the other in what we hope is an interesting and insightful way.

As we said, the Chandra and Toueg algorithm tolerates only crash failures and delays in the delivery of messages. Moreover, an important condition for the algorithm to work is that the number of non-crashed processes should always be smaller than the majority of processes in the system. The algorithm proceeds in rounds, and in many occasions the condition to advance to the next step is to have received a special message, or a majority of messages from other processes. Instead, from its informal description Paxos looks a much more powerful algorithm: it has a high tolerance to processes and communication faults and it is not tightly round based, which allows a certain freedom in message reception (for example, it is not mandatory for a process to receive a fixed number of messages in order to advance in the execution). However, the formalization of correctness proofs and the comparison of the two algorithms allows us to discover that Paxos is actually ”weaker” than the Chandra and Toueg algorithm, since it cannot give guarantees on Termination. In fact, if it is used in a lax way, the absence of strict conditions to advance in the execution of Paxos can lead to the generation of infinite runs.

This lack of the Termination property rises the question whether the Paxos algorithm can still be considered a Consensus algorithm. This is not simply a tedious philosophical question. In fact, if we imagine distributed systems as compositional structures, we can think of them as formed by layers of modules, where each module is like a black box that guarantees to provide certain specific properties. Thus, if a system architect wants to build an application using Consensus as one of the building blocks, he must trust that the module will provide the properties of Validity, Agreement **and** Termination. This is not the case if we use Paxos, since there might exist situations in which the module is not able to reach a final decision.

One surprising thing about the termination problem of Paxos is that some people know about it but they overlook it saying that in real environments the probability of non-termination is so remote that nobody should worry about it, while other people do not even realize the presence of this nuisance from the simple exam of the informal description and they look quite puzzled about our claim. Since this aspect of Paxos has never been clearly stated in literature, we think it is important to point it out and to clarify which parts of the algorithm can allow such undesired behavior. Our formalization has given us the means to easily do it.

To conclude, we would like to report some citations. A couple of months ago, while describing their experience in building a fault-tolerant database using the Paxos Consensus algorithm, Chandra, Griesemer and Redstone [10] stated that they encountered quite a lot of troubles in their job due to the enormous gap between theoretical pseudo-code specification of distributed algorithms (in the specific case Paxos) and the real representation. They think that ”[t]here are significant gaps between the description of the Paxos algorithm and the needs of a real-world system. In order to build a real-world system, an expert needs to use numerous ideas scattered in the literature and make several relatively small protocol extensions. The cumulative effort will be substantial and the final system will be based on an unproven protocol.” They also stated that their ”experience suggests that the gaps are non-trivial and that they merit attention by the research community”.

In this thesis, we provide a simple, yet formal way of describing and proving distributed algorithms. The system in which the algorithm is executed is represented as a composition of modules that support the algorithm. Any extension of the system can be represented as a new module that interacts with the algorithm and protocol extensions can be introduced as modifications of the algorithm description. The fact that a formal proof of the original algorithm already exists allows a quicker proof of the final expanded system, since for this it is enough to identify what parts of the proofs are influenced by the recent additions. Moreover, starting from our formalization, it is simple to implement a simulator of the algorithm that allows to get a first idea of its possible behavior and correctness.

Thus, we strongly hope that our work can constitute a step forward towards the closure of the gaps...

1.3 Contribution of the Thesis

In this section, we outline the contributions of this thesis through a list of the main results.

- We have proposed a simple way of formally describing and proving correctness of distributed Consensus algorithms using a transition system generated by inference rules.
- We have reported on the first formal proof of the Chandra-Toueg Consensus algorithm.
- We have applied our description and proof method also to the Part Time Parliament (a.k.a. Paxos) algorithm. Although there existed

already other formal proofs of correctness of Paxos, this work confirmed that it was possible to export our method also to other algorithms.

- We have proved that the amount of synchrony that is typically associated to Paxos (i.e., leader election) is not sufficient to guarantee its termination. Albeit many people claimed to be aware of it, we have not found in literature any clear trace of this result.
- We have described a condition and a strategy that allow a simplified version of Paxos to terminate with probability 1. This is not a crucial result, but it explores a possible way of reaching Termination that we believe should be expanded forward.
- We have used the common formalization method used to represent both algorithms to draw a comparison between the Chandra-Toueg and the Paxos algorithms, discovering that they are much more similar than what we could think by reading their informal descriptions. To conclude, we have discussed whether, on the basis of its non-respect of the Termination property, Paxos still deserves to be called ”Consensus algorithm” or not.

1.4 Overview of the Thesis

We now give a brief summary of the work presented in this thesis.

Chapter 2 is an introduction to distributed algorithms. We start by observing that the term distributed algorithms indicates a large variety of algorithms that span from trivial to very complex. We notice that the preferred way of expressing them is through pseudo-code or even plain natural language and we point out some ambiguities generated by such ways of description. We then focus on a quite complex, yet fundamental distributed problem: Consensus. Consensus is about succeeding in making a set of processes, that work in unreliable settings, decide on a common value. We explain in details what the problem is and what are the main strategies used to solve it.

Two of the most well known and widely used Consensus algorithms are the Chandra and Toueg algorithm and the Paxos algorithm. In Chapter 3 we present these two algorithms (more or less) as they can be found in the respective original papers, and we try to give informal arguments about their correctness.

In Chapter 4 we present the general idea of our formalization. In our approach, we distinguish between the states of the processes running the algorithm and the underlying modules that are needed in order to support the process actions. States and modules are then put together to form configurations. A configuration is obtained from another one by the application of some action and runs of the algorithm are presented as sequences of configurations.

The modules underlying the algorithm describe all the components of the system with which the process interacts during its execution. Three modules are needed by the Chandra-Toueg and the Paxos algorithms: Network, Reliable Broadcast and Failure Detectors (or Leader Election). Chapter 5 gives a general overview of the way we formalize these three modules.

After having set the bases, we can eventually give the formal description of the Chandra-Toueg (Chapter 6) and the Paxos (Chapter 7) algorithms. The structure of the two chapters is the same. This allows us to already remark that the two algorithms are pretty similar. There is one section in Chapter 6 that does not have the corresponding counterpart in Chapter 7. It is the formalization of module properties and state definitions that are needed for the proof of Termination of the Chandra-Toueg algorithm. A discussion on the issue of Termination in Paxos is provided later, in Chapters 10 and 11.

The following chapters present the correctness proofs for the Chandra-Toueg (Chapter 8) and the Paxos (Chapter 9) algorithms. Once again, both chapters proceed exactly in the same way up to the proof of Termination. Such property can be verified for the Chandra-Toueg algorithm, but not for Paxos. On the contrary: for Paxos we can provide a counter example of a never-ending run.

Chapter 10 presents a non-terminating run as the proof of a theorem stating that Paxos cannot guarantee the respect of the Termination property even in presence of ideal system conditions. We propose then an example of a special case in which a small modification in the Paxos algorithm would lead to its termination with probability 1.

The formalization work done in the previous chapters gives us the chance of writing a comparison between the Chandra-Toueg and the Paxos algorithms from the point of view of both the rules and the correctness proofs (Chapter 11). We remark that the biggest difference resides in the absence of Termination of Paxos due to a specific non-sequentiality choice made in the algorithm. This difference pushes us to the point of questioning whether Paxos can be considered a Consensus algorithm or not.

Chapter 12 presents some related work. First of all, we present a variant of the Failure Detectors representation we gave in Chapter 5. Then we explain how we tried a first formalization of the Chandra-Toueg Consensus algorithm using process calculi. After this account of our related work, we look at some of the main formalisms existing for the description of distributed algorithms. We explain how they can be used to represent and prove the algorithms and we briefly show differences and similarities with the approach presented in this thesis. We then present a brief excursus on History Variables and on the notion of Partial Synchrony. We end this Chapter by giving our opinion on the way algorithm formalization should be done and we specify why we think that process calculi are not yet ready to be used in this domain.

Finally, in the Appendix we present the description of a simulator that we have written starting from the Chandra-Toueg algorithm formalization and that can be useful to inspect some runs of the system.

List of Publications

Journal:

- R. Fuzzati and M. Merro and U. Nestmann. **Distributed Consensus, revisited**. *Acta Informatica*, 44(6), October 2007.

Conference:

- U. Nestmann and R. Fuzzati. **Unreliable Failure Detectors via Operational Semantics**. In Vijay A. Saraswat, editor, *Proceedings of ASIAN 2003*, volume 2896 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, December 2003.

- U. Nestmann, R. Fuzzati, and M. Merro. **Modeling Consensus in a Process Calculus**. In *CONCUR*, pages 393–407, 2003.

Book chapter:

- D. Bünzli, R. Fuzzati, S. Mena, U. Nestmann, O. Rütti, A. Schiper, and P. T. Wojciechowski. **Advances in the Design and Implementation of Group Communication Middleware**. Chapter of *Dependable Systems: Software, Computing, Networks*, volume 4028 of *Lecture Notes in Computer Science*, pages 172–194. Springer, 2006.

Workshop:

- R. Fuzzati and U. Nestmann. **Much ado about nothing?** *Electr. Notes Theor. Comput. Sci.*, 162:167–171, 2006.

Note: The work done formalizing and proving correctness of the Chandra-Toueg Consensus algorithm has been published in the paper ”Distributed Consensus, revisited” mentioned above. All the other results presented in this report are still unpublished work.

Chapter 2

Distributed Algorithms

The term *distributed algorithms* concerns a large variety of concurrent algorithms that can be employed for a wide range of applications. The term was originally used to refer to algorithms designed to run on many different processors ”distributed” over an extended geographical area. Nowadays, such term can also be used to refer to algorithms that run on local area network and even to algorithms for shared memory multiprocessors. Also, the term *distributed algorithm* can refer to a (relatively) big algorithm of which each process in the system executes one part with the purpose of splitting a task, or it can refer to a smaller algorithm that is executed by each process in the system with the purpose of achieving fault tolerance. The common trait of the algorithms that we define as distributed is to run on parallel processes that communicate with each other through some mean with the purpose of achieving a target. On the other hand, the attributes by which distributed algorithms might differ include the interprocess communication method, the timing model, the failure model, the addressed problem. Distributed algorithms are the answers to solve distributed problems, and there can be more than one algorithm that solve the same problem.

Communication The interprocess communicating method indicates in which way processes exchange information among themselves. For example, point-to-point communication means that there is a reserved channel for each pair of processes in the system. Such channels can then be qualified on the basis of their properties. For example, a channel is reliable if all messages that are sent are also received (if the receiver is correct), it is quasi-reliable if all messages sent from a correct process to a correct process are received, or it is lossy if some (or even all) messages that are sent are never received (even if the receiver is correct). Channels can also be replicating (or duplicating) if they replicate some (or even all) messages that pass through them. Another interprocess communication method that is rather peculiar is shared memory: processes communicate with each other by leaving messages into dedicated memory locations that hold office of mailboxes.

Timing The timing model refers to the way the system and the processes proceed. A system can be synchronous if there is an external clock that dictates the execution time for the processes and the transmission time for the communication means. In such a timing model it is very easy to detect message losses and process (crash) failures, and thus many distributed problems have a trivial solution. More interesting from the solution complexity point of view are asynchronous systems, where processes have each its own execution speed, that is unknown to the others and that can vary during time. Moreover, also the communication mean transmits messages with unpredictable timing, making it impossible for a process to tell whether another process is slow or crashed or whether a message is delayed or lost. As we will see later, there are problems for which there is no solution in such settings, and it is necessary to introduce at least a bit of synchrony (or a probabilistic component) in order to design a solution.

Failures The failure model refers to the way in which processes can fail to work as required. The simplest failure model is crash: the concerned process stops executing its tasks and becomes deaf and dumb, stopping sending and receiving messages. Following the failure model, a crashed process can stay as such forever or it can recover, restarting all its function but usually having lost memory of the past. Another failure model is Byzantine failure: the concerned process does not stop functioning, but starts behaving weirdly and sending incorrect messages. Usually such kind of failures are quite difficult to detect and to cope with.

As we have already hinted, the problems addressed by distributed algorithms are multiple and span from the cache coherence problem, ensuring integrity and consistency of data stored in local caches of a shared resource, to the Atomic Broadcast problem, where all clients of a system that contains replicated processes must receive a list of messages in the same order.

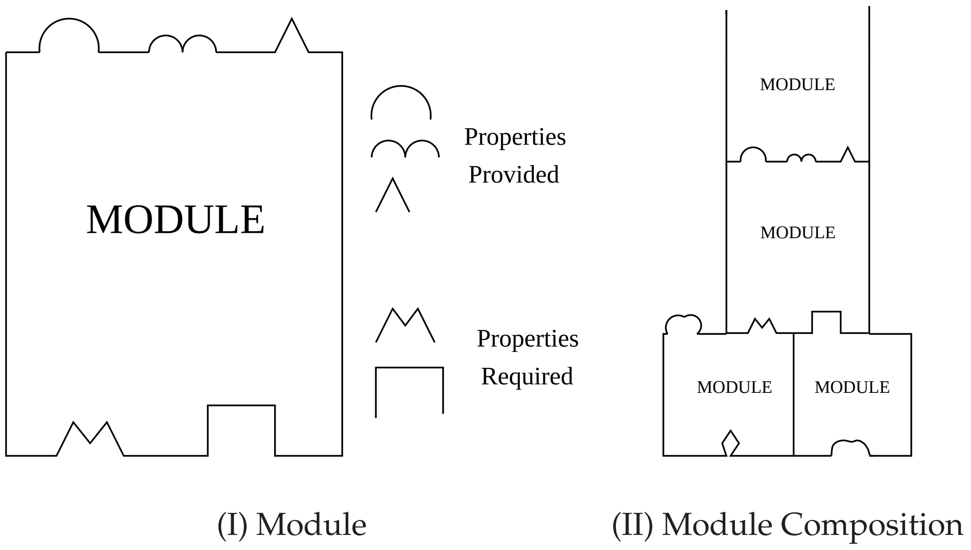

Figure 2.1: Modules and their composition

Modularity A non-negligible characteristic of distributed systems is that many algorithms rely, during their execution, on services that are in turn guaranteed by other distributed algorithms. For example, the most well known Atomic Broadcast algorithms can reach an agreement on the delivery order of a set of m messages because they execute m times a distributed algorithm that can make sure every process agrees with the others on one single message. Such algorithm, in turn, relies on another distributed algorithm ensuring that, if a message is sent from a process to all the others, eventually all (non-failed) processes receive the message. Moreover, even the reliability characteristics of the interprocess communication medium can be seen as services built on top of other completely unreliable mediums. This characteristic of dependence allows the engineers to treat each service as a module of which the ”inputs” and ”outputs” are known, and to compose the modules following their needs in order to create new services. (Figure 2.1).

2.1 Traditional Informal Descriptions

When learning about the compositionality characteristic of distributed algorithms, the reader has surely understood how important it is that each algorithm carefully provides the properties it declares, so that a user can concentrate on writing his/her solution to a distributed problem using already available algorithms as underlying layers, without having to worry about their correctness. Unfortunately, it is a tradition and a widespread habit in the distributed system community to express the algorithms in pseudo-code (in the best cases) or in natural language, and to give simple proof sketches to show that the algorithms satisfy the properties that they are required to provide.

We believe that this is not the best and safest way of ensuring that the algorithms and their implementations behave as expected and we fear that such lack of clarity in the specifications could lead to major accidents if the modules are used to build a critical application such as a flying controller for a plane or a nuke controller for a nuclear plant. First of all, pseudo-code can leave space to interpretation. For example, if we say

"process i sends message m to all processes"

do we mean that i sends m also to itself? If we read the sentence as it is, the answer is "of course, yes". But if we interpret the sentence by thinking that i already knows the content of m (since i is the one who has generated m) and can directly use it, then we obviously see no reason for i to send m also to itself. Another example is given by

"as long as process i is in state S it can send message m to any process".

What does *"any"* mean here? Can i send m many times to the same process and never to the others? Can i send m only to a small number of processes? When simply reading the rule, the intuitive answer is "yes". But if we consider the entire algorithm and its purposes, we see that it is pointless to waste messages by sending them to the very same process or only to a couple of processes and we agree that the sending of m should definitely be more targeted. Unfortunately, these little details can greatly affect the actions that are taken by i and by the other processes in the system, and one has to keep it in mind when designing, proving and implementing an algorithm. Moreover, proof sketches (especially when based on pseudo-code) can easily overlook improbable, but still possible, situations. For example, in round based algorithms for asynchronous systems, each process evolves in time and and in rounds. Since the system is asynchronous, processes do not have a global clock. Since a round is only a local concept, each process can advance as it pleases without waiting for the others, and the algorithm may reach configurations where each process is in a different round. Proofs of various distributed algorithm properties make heavy reference to the concept of round, but the relation between time and asynchronous rounds is never properly clarified. So, if we say that satisfying

Property X means that a process in round r receives (from other processes) a certain amount of messages related to the same round, then it is possible that, in time, Property X holds first for round r and subsequently for round $r-1$. An example of such a scenario is given in Figure 2.2. This is a non negligible detail that should not be overlooked when making a correctness proof.

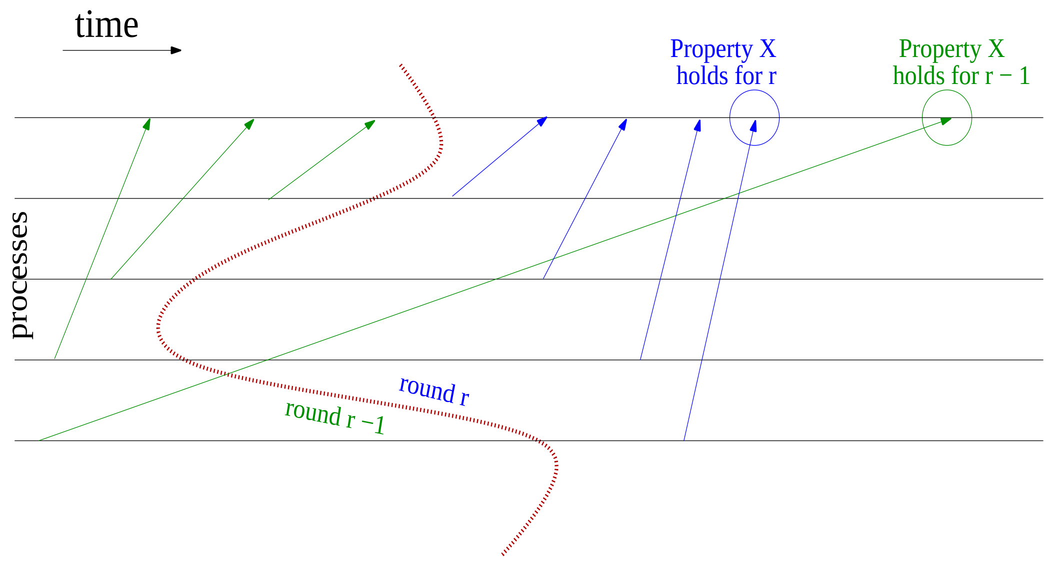

Figure 2.2: Time versus rounds in satisfaction of properties.

2.2 A Crucial Distributed Problem: Consensus

A distributed problem that is crucial for many applications and that we have studied it in great detail is Consensus. The purpose of Consensus is to ensure that all the (correct) processes of a system decide on a common value. It is used any time one wants to mask replication, that is any time a group of processes has to look like a single process to an external user. This is the reason why Consensus is used as a building block in many applications. For example, in the modular group communication middleware Fortika [9], that proposes a modular structure to allow a distributed group of processes to provide a reliable service in spite of the possibility of failures within the group, Consensus is placed right below the Atomic Broadcast module (Figure 2.3) and has to provide the latter with consistent and uniform messages.

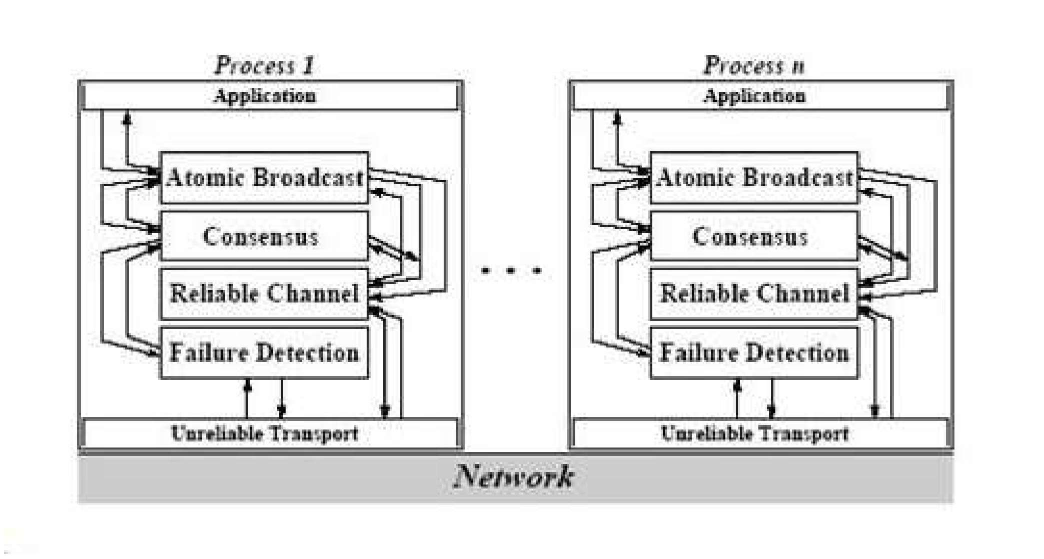

Figure 2.3: Building blocks in Fortika

Consensus is also used as a test to classify wait-free implementations of concurrent data objects [28]. A wait-free implementation guarantees that any process in a system can complete any operation in a finite number of steps, regardless of the execution speed of the other processes. Consensus is used in such a context to derive a hierarchy of objects: each object has an associated *consensus number* that is the maximum number of processes for which the object can solve the Consensus problem and that is used to put the objects into a relation of order. The hierarchy is then used to show impossibility results such as "it is impossible to construct a wait-free implementation of an object with *consensus number* n from an object with a lower one". Moreover, since Consensus is considered to be universal (it can solve Consensus for any number of processes) it has been proved that any other shared object can be implemented on top of a system that solves Consensus.

The Consensus problem is characterized by three properties that every algorithm solving Consensus has to satisfy and that are typically formulated as follows:

1. If a process decides a value v, then v was proposed by some process (*Validity*).
2. No two correct processes decide differently (*Agreement*).
3. Every correct process (eventually) decides some value (*Termination*).

As we can see, two of these properties, Validity and Agreement, are related to the fact that ”nothing bad can ever happen”, this is what is usually called *Safety*. The Termination property guarantees instead that ”something good will eventually happen”, and this is what is called *Liveness*. Even though, as we will see later in this thesis, some authors think that Safety is the most important property of a system, in our opinion Liveness should not be underrated since there might be modules or applications that wait for the algorithm to give them an answer, and therefore depend on it to be able to proceed.

While solving Consensus in synchronous settings is a trivial task even in presence of failures, things get far more complicated when trying to solve Consensus in an asynchronous system. So complicated that the FLP Result tells us that it is impossible to solve Consensus in an asynchronous system if even only one single process can crash. In fact, in their seminal paper [21], Fischer, Lynch and Paterson have proved that, in a crash-failure model, it is impossible to simultaneously satisfy the three properties above if some process may fail. More precisely, they prove that, under the assumption of Validity and Agreement, the possibility of even a single process crash invalidates Termination: there would always be a non-terminating system run, where no decision is reached. Thus, while the properties of Validity and Agreement can be guaranteed through appropriate algorithms, in order to respect Termination it is necessary to introduce some notion of *synchrony* into the system. Related to the degree of synchrony and the failure model is also the number of failures that a system should allow if we want Safety to be guaranteed. For example, to ensure Agreement for crash failures in asynchronous systems we need to have a strict majority of correct (non-crashed) processes. If this is not the case, we can partition the system in two or more sets of processes where Agreement is reached among processes inside the same set, but not among different sets. Figure 2.4 represents the case of a system with five processes that is partitioned, probably due to network failure between two areas. If we build the algorithm in such a way that processes do not need to wait for a majority of answers from other processes before deciding, we can have the top processes suspecting the bottom processes of having crashed (or at least of being cut apart) and thus deciding for value 1, while the bottom processes suspect the top processes of having crashed thus, not waiting for a majority of messages, deciding for value 0. In Sections 8.2.2 and 9.2.2 we will see clearly how and where this ”majority hypothesis” is used in the proof of Agreement. For the moment we can intuitively say that, since two majorities have a non-empty intersection, from one round to the following one there is always at least one process of the new majority that acts as a spokesman of the previous majority (Figures 2.5 and 2.6).

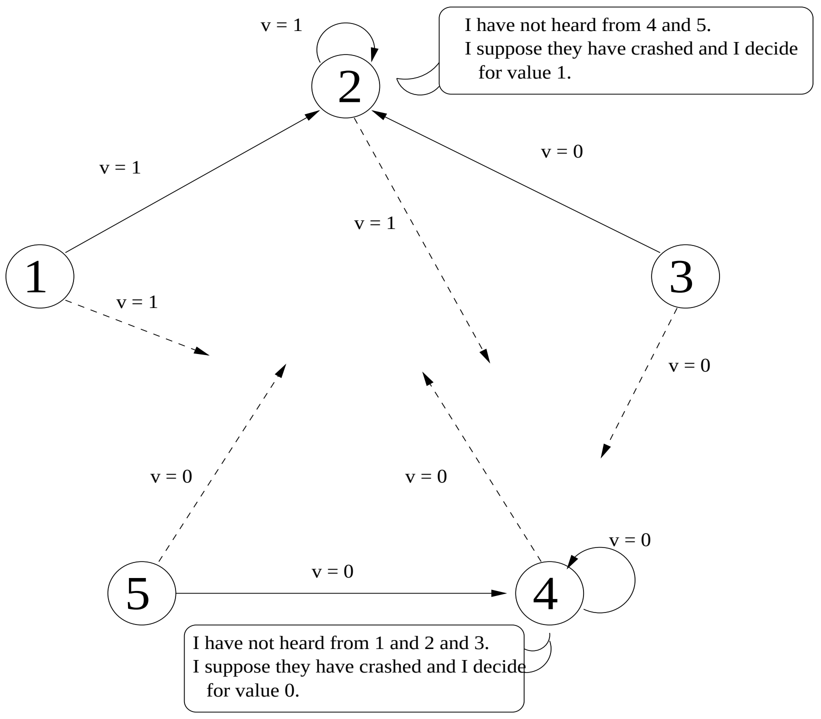
v = 1
I have not heard from 4 and 5.
I suppose they have crashed and I decide for value 1.
2
v = 1
v = 0
v = 1
1
3
v = 1
v = 0
v = 0
v = 0
v = 0
5
v = 0
4
I have not heard from 1 and 2 and 3.
I suppose they have crashed and I decide for value 0.

Figure 2.4: Violation of Agreement if No Correct Process Majority

2.2.1 Algorithms

Many algorithms have been proposed to solve Consensus, and the choice on which one to use depends on the system settings (interprocess communicating method, timing model, failure model) and on the desired performance of each particular case. Also, some algorithms allow the process to stop once Consensus has been reached, while others go on indefinitely. Once a Consensus algorithm that is suitable for the needs has been chosen, the algorithm is given in identical copies to every process in the system and each process executes the algorithm entirely. Usually, algorithms that solve Consensus proceed in *rounds*. A *round* is a macro-step of an algorithm that has to be repeated, either until the target is reached or indefinitely, and describes a sequence of steps that the process has to execute. Each round is characterized by a sequential number and such number is usually inserted in messages in order to be able to put them into a context. Given the FLP Result mentioned above, to solve Consensus in an asynchronous system with failures it is necessary to introduce additional components to the system. These components can be randomization, like in [4], or devices capable to add degrees of synchrony, like in [11, 12, 35],

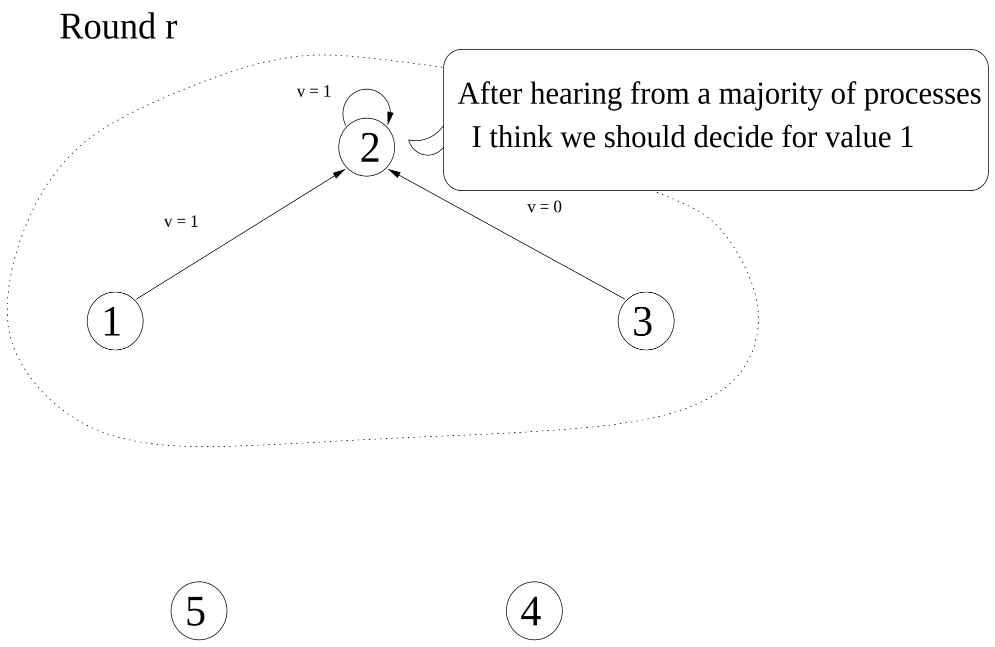
Round r
After hearing from a majority of processes I think we should decide for value 1
v = 1
v = 1
v = 0
1
2
3
4
5

Figure 2.5: Majority at Round r

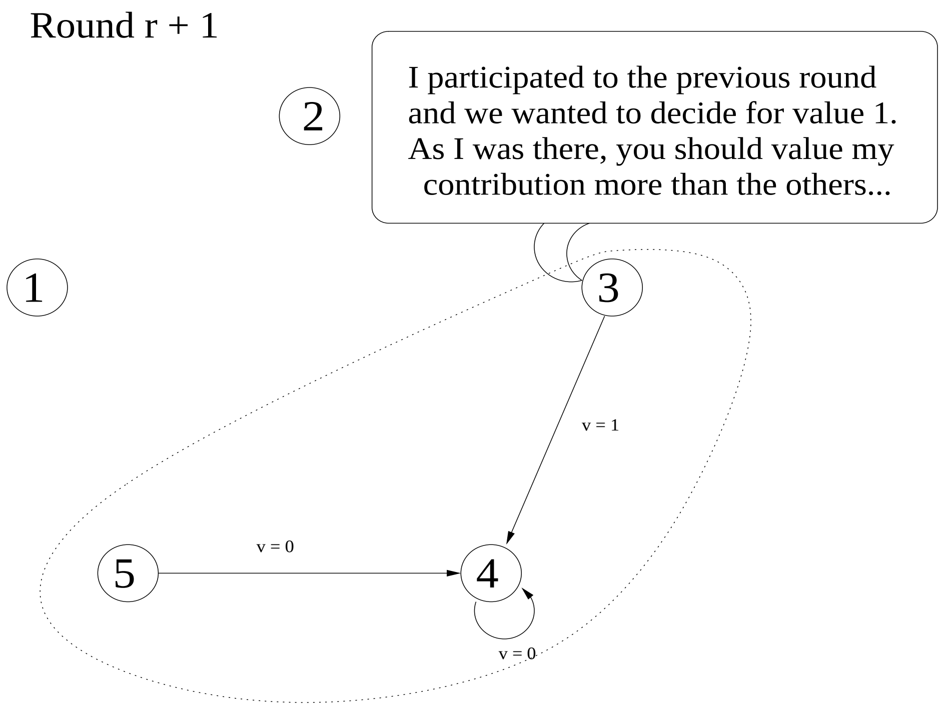
Round r + 1
I participated to the previous round and we wanted to decide for value 1. As I was there, you should value my contribution more than the others...
v = 1
v = 0
v = 0
1
2
3
4
5

Figure 2.6: Majority at Round r+1

or even a combination of the two [2].

Randomized Algorithms

The Ben-Or's randomized Consensus algorithm [4] is the most well known algorithm that can solve Consensus in asynchronous systems and in presence of crash failures. The algorithm goes as shown in Table 2.1 where n represents the number of processes in the system, x_i indicates process i current estimate of the decision value, v_i is the initial value of process i, r is the round number and R and P indicate the two possible phases inside a round. The algorithm is designed to work in asynchronous settings, with reliable point-to-point channels and crash failures. It is assumed that only a number f of processes, with $n > 2f$, can crash. All exchanged messages carry the indication of the phase and the round number in which the sending process is at the time of creation of the message. At the beginning of each round, each process sends a message containing its estimate, then it waits for $n - f$ messages from the other processes and proceeds to phase P. If at least $n/2$ of these messages carry the value v, then the process sends a message containing value v to all other processes, otherwise it sends a message indicating that it does not know a good value $?$, then it waits for $n - f$ messages from the other processes. If it receives at least $f + 1$ messages carrying the same value (different from $?$) then the process decides for such value. If it receives at least one message carrying a value (different from $?$), then it changes its estimate x for the new value v. But if it receives only messages saying that there is no good value, then it asks a randomized number generator to return a number that is either 0 or 1 and changes its estimate for such a random value.

The idea behind this algorithm is that, since each process always wait for $n - f$ messages before taking any action and any decision, the safety properties are respected, and it is not possible to identify two sets of processes in the system that can take two different decisions. As for Termination, the fact that, at any round, the probability of having a majority of processes proposing the same value is greater than zero implies that the probability of non terminating during a certain round is bounded. Since the probability of non terminating in a round is independent of the probability of non terminating in another, the fact that the non termination probability is bounded in each round implies that the probability of termination of the algorithm goes to 1. It is important to underline that this algorithm does not guarantee the Termination property, but it ensures that the algorithm terminates (which in this case means that all non crashed processes execute action *decide(v)* for some value v) with probability 1.

Every process executes the following:

```
PROCEDURE RANDOMIZED-CONSENSUS $(v_i)$
  $x_i \leftarrow v_i$
  $r \leftarrow 0$
  while true
  {  $r \leftarrow r + 1$
     send $(R, k, x)$ to all processes
     wait for messages of the form $(R, r, *)$ from $n - f$ processes
     if received more that $n/2$ $(R, r, v)$ with the same $v$
        then send $(P, r, v)$ to all processes
        else send $(P, r, ?)$ to all processes
     wait for messages of the form $(P, r, *)$ from $n - f$ processes
     if received at least $f + 1$ $(P, r, v)$ with the same $v \neq ?$
        then decide(v)
     if received at least one $(P, r, v)$ with $v \neq ?$
        then $x_i \leftarrow v$
        else $x_i \leftarrow 0$ or $1$ randomly
}
```

Table 2.1: The Ben Or Randomized Consensus Algorithm

Randomized algorithms tend to be inefficient in practical settings. Moreover, their performance is usually more related to luck (many processes starting with or drawing the same random bit) than to how well behaved the system is (absence of failures or messages delivered in timely fashion).

Failure Detectors and Leader Election

A second strategy to overcome the FLP impossibility result is to attach to each process in the system a special module capable of making ”good guesses” on which processes are alive and which are crashed. At any moment in time, this module makes available for its partner process a list of processes that are suspected to have crashed. The process can then use such information to take decisions on whether to wait for a message from another process or not. Clearly, it is pointless to wait for a message that should have been sent from a process that is suspected to have crashed, and it is in the best interest of the whole system to proceed further without waiting for too long. On the other hand, if the process that should have sent the message is detected as being alive, then it is pointless not to wait for its message because that would force the system to go through a new round, thus wasting time and resources.

		Accuracy			
		Strong	Weak	Eventual Strong	Eventual Weak
Completeness	Strong	*Perfect* $\mathcal{P}$	*Strong* $\mathcal{S}$	*Eventually Perfect* $\diamond\mathcal{P}$	*Eventually Strong* $\diamond\mathcal{S}$
	Weak	$\mathcal{Q}$	*Weak* $\mathcal{W}$	$\diamond\mathcal{Q}$	*Eventually Weak* $\diamond\mathcal{W}$

Table 2.2: Taxonomy of Failure Detectors

Chandra, Hadzilacos and Toueg [11, 12] have called these modules *failure detectors* [1], and have proposed a taxonomy (see Table 2.2) to classify them based on their characteristics with respect to completeness and accuracy of suspicions. Completeness addresses crashed processes that should be suspected, while accuracy addresses correct processes that must not be

[1]Sometimes we will refer to them as FDs.

suspected. Chandra, Hadzilacos and Toueg distinguish two kinds of completeness and two kinds of accuracy, which combined together give four different classes of failure detectors. They are defined as follows:

Strong Completeness
Eventually *every* process that crashes is permanently suspected by *every* correct process.

Weak Completeness
Eventually *every* process that crashes is permanently suspected by *some* correct process.

Strong Accuracy
No process is suspected before it crashes.

Weak Accuracy
Some correct process is suspected.

Since accuracy addresses correct processes that must not be suspected, the two accuracy properties can be relaxed making them become *eventual*:

Eventual Strong Accuracy
There is a time after which *correct* processes are *not suspected* by any correct process.

Eventual Weak Accuracy
There is a time after which *some* correct process is never suspected by *any* correct process.

In [12], Chandra and Toueg proved that it is actually possible to transform any given failure detector that satisfies weak completeness into a failure detector that satisfies strong completeness, thus simplifying the taxonomy only to the first row of Table 2.2.

In [11, 12], Chandra, Hadzilacos and Toueg proved that the weakest failure detector that can be used to solve Consensus in a crash failure system where a majority of processes is not faulty has to guarantee the properties of (Strong) Completeness and Eventual Weak Accuracy. Such failure detector is called $\diamond\mathcal{S}$ and it eventually provides its partner process with a list of all processes that are crashed (Strong Completeness). Moreover, all failure detectors in the system (at least those whose partner process has not crashed) eventually ”trust” the very same process, i.e., they all share the belief that this particular process has not crashed, so its name is eventually never present in any of the suspicion lists (Eventual Weak Accuracy).

Always in [11] Chandra, Hadzilacos and Toueg also proved that $\diamond\mathcal{S}$ is actually equivalent to another type of failure detector called Ω. Instead of giving to its partner process a list of processes that are suspected to have crashed, Ω outputs only the name of a process that it believes to be correct. Processes are then left free to take actions against any process but the ”trusted” one, and the requirement for Ω is simply that, eventually, all failure detectors in the system trust the same process. Thus, when using Ω as failure detector, processes can go through periods when they trust different processes, but they always eventually come to trust the same one.

Another concept that has been used for a long time when trying to overcome the FLP impossibility result is that of Eventual Leader Election. A leader is a correct process that is recognized by other processes as the only one being allowed to perform some special actions (typically coordinating the interactions between the participants). The Eventual Leader Election property states that there is a time after which all correct processes recognize the same correct process as the leader. Several leaders can coexist during an arbitrarily long period of time, and there is no way for the processes to learn when such ”anarchy” period is over, but eventually only one process is unanimously recognized by all the (correct) processes as being the leader. For a relatively long time, those of Failure Detection and Eventual Leader Election have been regarded as two different concepts. Now we know that they represent exactly the same abstraction, and if we look at Ω we realize that what it does is exactly ”electing a leader”: the process names that each Ω failure detector outputs to its partner process are the leaders, and when all Ω failure detectors eventually output the same process name that is the ”election” of the eventual unique leader.

Thus, in the remaining of this thesis, we will often use the terms Failure Detection and Leader Election as synonyms.

Hybrid Algorithm

Since both randomization and failure detection have their advantages and disadvantages, Aguilera and Toueg [2] proposed to combine them in an hybrid algorithm that is mainly based on failure detectors but that turns to randomization to ensure Termination if many failures occur or if messages take longer than expected to be delivered. In fact, Aguilera and Toueg start from the observation that, most of the time, systems are more or less correct and thus failure detectors are extremely efficient and reliable. Problems with failure detectors arise when they erroneously believe that processes have crashed due to delays in the network. In such cases, the algorithm becomes randomized and ensures that, at least, Termination will be reached with probability 1.

Chapter 3

The Chandra-Toueg and the Paxos Consensus Algorithms

Two of the most well known and exploited Consensus algorithms are the Chandra and Toueg algorithm [1] and the Part Time Parliament algorithm, also (or better) known as Paxos. Both these algorithms are conceived to work in asynchronous message passing systems, making use of failure detection/leader election.

We have started our formalization work on the Chandra-Toueg algorithm because it was the one used for the Consensus building block inside Fortika [9] and we wanted to be 100% sure that such block was respecting its specifications. Once we proved correctness for CT [24], we imagined a possible future scenario where Fortika could be employed, so we thought of a system with crashes and recoveries of processes and with unreliable communications. That pushed us to look at Paxos and to do on it the same work of formalization as we did so successfully for CT.

3.1 The Chandra - Toueg Consensus Algorithm

In this section, we explain the details of the Consensus algorithm proposed by Chandra and Toueg in [12]. Such algorithm works in asynchronous systems with crash failures, and its correctness depends on the availability and correct functioning of three lower-level services: quasi-reliable point-to-point communication, reliable broadcast and the FD $\diamond\mathcal{S}$.

We recall here that, for quasi-reliable point-to-point communication (we will often speak of QPP), each pair of processes in the system is con-

[1]Sometimes we will refer to the Chandra and Toueg Consensus algorithm as the Chandra-Toueg algorithm or more simply as CT.

nected through a reserved bidirectional channel with the property that all messages sent from a correct process to a correct process are eventually received. We recall also that the FD $\diamond\mathcal{S}$ satisfies the properties of Strong Completeness (eventually every process that crashes is permanently suspected by every correct process) and Eventual Weak Accuracy (there is a time after which some correct process is never suspected by any correct process).

The algorithm proceeds in *asynchronous rounds*: each of the n processes counts rounds locally and advances independently in the execution of the algorithm. Furthermore, the algorithm is based on the *rotating coordinator* paradigm: for each round number r, a single process is predetermined to play a coordinator role, while all other processes in the same round play the role of *participants*. Let n be the number of processes, then the function $\mathrm{crd}(r)$ returns the integer $((r-1) \mod n)+1$ denoting the coordinator of round r. This function is known to all the processes, therefore each of them is able to compute at any time the identity of the coordinator of a round.

Each of the n processes stores a local *estimate* of the sought decision value, together with a *stamp* that tells when—i.e., in which round—the process has come to believe in this estimate. We will refer to the pair (*estimate*,*stamp*) as a *belief*. The goal of the algorithm is to reach a situation in which a majority of processes agree on the same estimate. To reach such a situation, coordinators and participants exchange their current beliefs, round after round, through QPP-messages. The role of a coordinator is (i) to collect sufficiently many current beliefs, (ii) to find out the most recent estimates by comparing the respective round stamps, (iii) to select one estimate among those, and (iv) to disseminate its selection to the participants such that they can update their local beliefs accordingly.

3.1.1 The Algorithm in Pseudo-code Format

We present here the Chandra and Toueg algorithm expressed in pseudo code. This representation is not completely identical to the original algorithm [12], since we have chosen to apply some small changes in order to ease the formalization step. The differences between the two versions of pseudo code are mainly due to the data structures involved. A more detailed analysis is given at the end of this section, where we will have all the knowledge that is necessary to understand that such differences are absolutely negligible.

Keywords. We use the programming constructs `while` and `if·then·else·` with their standard interpretation. We also use the construct `await` to indicate that execution is suspended until some associated boolean condition is satisfied, and we use `when` to indicate that the execution of some associated primitive triggers the execution of the subsequent code.

Data structures. Each process carries a state containing the following three ingredients:

- 'counter': storing the local round number,
- 'belief': storing the pair (or binary record) of 'estimate' and 'stamp', and
- 'decision': to control the exit condition of a `while`-loop.

We consider process states as arrays of records. When we are interested in selective extraction and update, we treat a process state explicitly as a record with the labels indicated above. So, we use $x.\mathrm{label}$ to denote record selection; we use $x \leftarrow v$ to denote assignment of all the record fields of x, and $x.\mathrm{label} \leftarrow v$ to denote assignment of label of x.

Program structure. The pseudo code of Table 3.1 describes the behavior of any process i, for $i \in \mathbb{P}$. Any process i locally starts a Consensus run by executing C_PROPOSE(v_i), where v_i is taken from a set $\mathbb{V}$ of proposable values. The process first initializes its local state: the 'counter' and the 'stamp' are initialized to zero, the 'estimate' is initialized to v_i, the 'decision' is left undefined, written $\perp$. Then the process enters a `while`-loop. When starting a Consensus run, the process launches also a concurrent sub-routine that can modify the value of the state component 'decision', execute C_DECIDE and prevent the `while`-loop from restarting.

Each cycle through the `while`-loop represents a round. It proceeds in steps (labeled `P1`...`P4`) that represent the four phases of a round. But before explaining in detail the structure of the `while`-loop, let us have a look at QPP-messaging and Reliable Broadcast, since they are heavily exploited inside a round.

QPP-messaging. In QPP_SEND $(\mathit{sender}, (\mathit{round}, \mathit{phase}), \mathit{contents})$ TO *receiver* the pair $(\mathit{round}, \mathit{phase})$ plays the role of a sequence number, which, together with the *sender* and *receiver* identification, allows us to distinguish—regardless of the *contents*—all QPP-messages ever sent during a Consensus

```
PROCEDURE C_PROPOSE $(v_i)$
  $state$.counter $\leftarrow 0$
  $state$.belief.value $\leftarrow v_i$
  $state$.belief.stamp $\leftarrow 0$
  $state$.decision $\leftarrow \bot$

  while $state$.decision $= \bot$
  {  $state$.counter $\leftarrow$ $state$.counter $+ 1$
     $r \leftarrow$ $state$.counter

     P1   QPP_SEND $(i, (r, \mathsf{P1}), state.\mathrm{belief})$ TO crd$(r)$

     P2   if     $i = \mathrm{crd}(r)$
          then  {  await  $|\ \mathrm{received\_QPP.P1}^r\ | \ \geq\ \lceil (n{+}1)/2 \rceil$

                   for $1 \leq j \leq n$
                   do  QPP_SEND $(i, (r, \mathsf{P2}), \mathrm{best}(\mathrm{received\_QPP.P1}^r))$ TO $j$
                }

     P3   await  (  $\mathrm{received\_QPP.P2}^r \neq \emptyset$  or  suspected(crd$(r)$)  )
          if     $\mathrm{received\_QPP.P2}^r = \{(\mathrm{crd}(r), (r, \mathsf{P2}), (v, s))\}$
          then  {  QPP_SEND $(i, (r, \mathsf{P3}), \mathtt{ack})$ TO crd$(r)$
                   $state$.belief $\leftarrow (v, r)$
                }
          else     QPP_SEND $(i, (r, \mathsf{P3}), \mathtt{nack})$ TO crd$(r)$

     P4   if     $i = \mathrm{crd}(r)$
          then  {  await  $|\ \mathrm{received\_QPP.P3}^r\ | \ \geq\ \lceil (n{+}1)/2 \rceil$
                   if    $|\ \mathrm{ack}(\mathrm{received\_QPP.P3}^r)\ | \ \geq\ \lceil (n{+}1)/2 \rceil$
                   then  RBC_BROADCAST$(i, state.\mathrm{belief})$
                }
  }


when  RBC_DELIVER$(j, d)$
{  if     $state$.decision $= \bot$
   then   $state$.decision $\leftarrow d$
          C_DECIDE ($state$.decision.value)
}
```

Table 3.1: The Chandra-Toueg Consensus via Pseudo Code

run, under the simple condition that every process sends to any other process at most one message per $(round, phase)$. The type of *contents* depends on the respective *phase*: it may be a belief (phase P1 or P2) or an `ack`/`nack` (phase P3). Note that the algorithm never sends QPP-messages in phase P4.

A QPP-message travels through the network from the sender to the receiver. By the assumption of quasi-reliability, the reception is guaranteed if neither the sender nor the receiver crash. More precisely, if the sender crashes after sending, then the message may still be received at some point, but there is simply no guarantee for this event to ever happen. One might expect the pseudo code to feature explicit executions of QPP_RECEIVE to describe individual receptions. However, since our pseudo code often blocks until certain messages have arrived, and since it sometimes acts if a particular message has not yet arrived, it makes sense to silently add incoming messages, on reception, to some local *message buffer* (implicitly defined). In order to express the fact that the algorithm waits for a particular condition on the message buffer to become true, our pseudo code features the use of an `await` construct. As appropriate buffering data structures we introduce sets denoted by received_QPP, for which we conveniently also provide the notation received_QPP.P^r that extracts the received messages, according to *round* $r \in \mathbb{N}$ and *phase* $P \in \{\mathsf{P1}, \mathsf{P2}, \mathsf{P3}\}$. Note that each process i maintains its own collection of such sets, and that no process j can access the collection of another process i.

Reliable Broadcast. Reliable Broadcast is a special communication abstraction that gives more guarantees than simple QPP-messaging. In particular, it ensures that either all correct processes get (in this case we speak of *deliver*) a message, or none of them does. Moreover, if the sender of a message is a correct process, then such process eventually delivers the message to itself, thus obliging all the other correct processes to do the same. In contrast to QPP-messaging, where we actively await the arrival of suitable messages of certain kinds, no receiver-side buffering is required for the treatment of RBC_DELIVER. Thus, in the Chandra and Toueg algorithm, the reception of broadcast messages is performed individually and directly using `when`-clauses.

Loop structure. As said before, every process goes through four phases per round, i.e., per `while`-cycle. In the following, when we speak of a process executing an action, we implicitly mean a correct process.

In phase P1, each process sends to the coordinator its current belief, i.e., a pair consisting of an estimate and its stamp.

In phase P2, all the processes except for the coordinator move immediately to phase P3. The coordinator waits until it receives sufficiently many proposals (here, we use the standard notation $|\cdot|$ to denote the cardinality of sets), namely at least $\lceil (n{+}1)/2 \rceil$, and selects among them the one with the highest 'stamp'. Here, $\mathrm{best}(\mathrm{received\_Q{\small PP}}.\mathtt{P1}^r)$ indicates a (local) function that performs this operation. This "best" proposal is called the *round proposal* and is distributed to all participants, again using QPP-messages. The coordinator then also moves to phase P3.

In phase P3, each process waits for a moment in which either the coordinator's round proposal has arrived at the local site or the coordinator is suspected by the process's (local) FD. In both cases, the process then sends an acknowledgment (positive: `ack` or negative: `nack` respectively) to the coordinator and proceeds to the next phase. Only in the positive case, the process adopts the round proposal. Recall that, by our definition of FD, a process (and in particular the coordinator) cannot suspect itself. As a consequence a coordinator will always adopt the round proposal and send a positive acknowledgment to itself.

In phase P4, all the processes except for the coordinator proceed to the next round by `while`-looping. The coordinator waits until it receives sufficiently many acknowledgments. Here, $\mathrm{ack}(\mathrm{received\_Q{\small PP}}.\mathtt{P3}^r)$ selects the subset of $\mathrm{received\_Q{\small PP}}.\mathtt{P3}^r$ that contains all the received messages carrying a positive acknowledgment. If the cardinality of $\mathrm{ack}(\mathrm{received\_Q{\small PP}}.\mathtt{P3}^r)$ is at least equal to $\lceil (n{+}1)/2 \rceil$ then we say that the value proposed by this majority is *locked* and the coordinator "initiates a decision", otherwise the coordinator fails this occasion and simply ends this round. In both cases, the coordinator then proceeds to the next round by re-entering the `while`-loop.

The above-mentioned "initiating a decision" means to execute RBC_BROADCAST$(i, \mathit{state}.\mathrm{belief})$, by which the round proposal that has just been adopted by a majority of processes, is transmitted to *all* processes. The corresponding receptions RBC_DELIVER(j, d) are captured by the `when`-clause that is running concurrently: processes may RBC_DELIVER regardless of their current round and phase. On execution of RBC_DELIVER(j, d), and only when it happens for the first time, the process "officially" announces its decision for the delivered value by executing C_DECIDE. Although it is not explicitly indicated by Chandra and Toueg, the `when`-clause must be performed atomically to guarantee Termination. Notice also that, by the usual convention on Reliable Broadcast, also the broadcast initiator must itself explicitly perform the delivery before it can decide. Since Reliable Broadcast satisfies the Agreement property, every non-crashed process will eventually receive the broadcast message.

Differences to the version of Chandra and Toueg. Apart from slightly different keywords, our pseudo code deviates from the original one in the following minor ways:

- We chose to compact all variables that contribute to the state of a process into a single data structure;
- We chose a more abstract primitive to indicate process suspicion (thus closely following the idea of module composition expressed in Chapter 2), while the original code directly referred to the author's mathematical model of FD;
- We use the FD Ω instead of the original $\Diamond\mathcal{S}$ (but we have seen is Section 2.2.1 that they are equivalent);
- We use explicit proper data structures (instead of leaving them implicit) to deal with received QPP-messages, to select the subsets $\mathrm{ack}(\mathrm{received\_QPP}.P^r)$ for $P \in \{\mathsf{P1}, \mathsf{P3}\}$, and we also introduce the operation $\mathrm{best}(\mathrm{received\_QPP}.\mathsf{P1}^r)$ to operate on these structures.

3.1.2 An Intuitive Argument of Correctness

Intuitively the algorithm works because coordinators always wait for a majority of messages before they proceed (which is why, to ensure the computation is non-blocking, strictly less than the majority are allowed to crash).

Validity holds because none of the clauses of the algorithm invents any value. Whenever a value is written, it is always *copied* from some other data structure, or selected among several such sources. Obviously then, all values that occur in a reachable system configuration must originate from some initially given value.

According to Chandra and Toueg, Agreement and Termination can intuitively be explained by means of a three-epoch structure of runs. In epoch 1 everything is possible, i.e., every initial value might eventually be decided. Moving to epoch 2 means that a value *gets locked*, i.e., for some round r (actually: the smallest such) a majority of processes send a positive acknowledgment to the coordinator of r. Since the coordinator uses a maximal time-stamp strategy when choosing its proposal, and since only a minority of processes may crash, a locked value v will always dominate the other values: should some value get locked in a greater round, then it will be the very same value v that was previously locked. In epoch 3 all (correct) processes decide the locked value. Due to the Reliable Broadcast properties, if a correct process decides, then all other correct processes decide as well.

Agreement is a safety property which guarantees that nothing goes wrong. In this case it holds because (1) if two values v_1 and v_2 are locked in two different rounds, then $v_1 = v_2$; (2) if a process decides, it always decides on a locked value. Termination is a liveness property which guarantees that something good eventually happens. In this case it holds since the FD properties enforce the eventual transition of the system from epoch 1 to epoch 2, and from epoch 2 to epoch 3.

The proof of Termination relies on the combination of two sub-results (1) *correct processes do not get blocked forever*, and (2) *some correct process eventually performs some crucial action*. As to (1), the requirement that only a minority of processes can crash ensures that *coordinators do not block forever* when waiting for a majority of messages (in phases P2 and P4); moreover, the Strong Completeness property of $\diamond\mathcal{S}$ ensures that all the crashed processes will be eventually suspected; in particular *no process can block forever waiting for a crashed coordinator*. As to (2), the Eventual Weak Accuracy property ensures that eventually *some correct coordinator will be no longer suspected*, will receive a majority of positive acknowledgments, and will RBC-broadcast the best value; the RBC-properties then ensure that all values that are RBC-broadcast by coordinators will eventually be RBC-delivered (and hence decided) by all correct processes.

3.2 The Paxos Consensus Algorithm

The Part Time Parliament, commonly called Paxos, is a Consensus algorithm that has been proposed by Lamport [35, 36] to ensure that all the correct processes of an asynchronous system can decide on the same value. More precisely, the Paxos algorithm is designed to ensure the properties of Agreement (all processes that decide, decide for the same value) and Validity (the value that is decided on is the initial value of one of the processes) even in presence of lossy and duplicating channels and crash/recovery failures. As we will explain in more details in Chapter 10, we cannot really say that Paxos is a ”real” Consensus algorithm because there are no (or, at least, we could not find) ways of giving formal conditions other than eventual synchrony to achieve the Termination property (all the correct processes eventually decide on a value) that is required by the definition of consensus. We can therefore affirm that Paxos is an algorithm capable of ensuring that ”nothing bad happens” and that no process ever makes a decision different from the one made by another process, even in presence of very unreliable network and crash/recovery failures.

The algorithm proceeds in asynchronous *ballots* that, for the sake of uniformity with CT, we will call *rounds*. Paxos is not based on the rotating coordinator paradigm as CT, but on the election of one (or possibly multiple) leader(s). Each process has an infinite set (disjoint from the ones of the other processes) of round numbers that can sequentially be used when the process becomes leader. Processes that are not leaders play the role of participants. Since it is the leader that proposes the round number, participants cannot predict, or compute as in CT, the round evolution, but are simply invited by leaders to participate to rounds carrying higher numbers.

Each of the n processes stores a local *estimate* of the sought decision value, together with a *stamp* that tells when—i.e., in which round—the process has come to believe in this estimate. As in CT, we refer to the pair (*estimate*,*stamp*) as a *belief*. The goal of the algorithm is to reach a situation in which a majority of processes agree on the same estimate. To reach such a situation, leaders and participants exchange their current beliefs, round after round, through network messages. The role of a coordinator is (i) to collect sufficiently many current beliefs, (ii) to find out the most recent estimates by comparing the respective round stamps, (iii) to select one estimate among those, and (iv) to disseminate its selection to the participants such that they can update their local beliefs accordingly.

3.2.1 The Algorithm in Natural Language Format

In [35] the Part Time Parliament algorithm is described in natural language, sometimes using Greek fonts that make things look quite cryptical. The appendix contains instead a more formal description, that still leaves some doubts as to what each action is supposed to do. For example, we find an action ”Send a NextRound message to any participant”, but it is not specified how many times this action should take place nor it is specified whether, in case of multiple sending, the receiver should be a different one. This leaves space for interpretation and a not too smart user could even understand that he is allowed to infinitely send the same message to the same process without sending it to anybody else.

We rewrite here the Paxos basic protocol as we understand it from a combination of the natural language and the ”formal” description in [35].

We take the liberty of changing some of the parameter'names in order to come closer to our forthcoming representation of the algorithm.

First of all, we have to explain the structure of each process. Each participant to the algorithm has a record with four fields that can never be deleted, but only updated with more recent values. The first field stores the last *request number* that the process proposed to the others. The second field stores the last *prepare request number* to which the process has answered. The third field stores the *belief* of the process, consisting of a value (belonging to a set $\mathbb{V}$ of possible values) and a number that represents the last prepare request number in which the process updated the field. The fourth field stores the *decision*: the value that the process adopts as its final answer for the algorithm.

We now have all the elements to write how a "round" of the Paxos algorithm evolves.

1. Process i, that is leader, chooses a new request number *next* that is greater than the last request number it has tried. It sets its request number field to *next* and sends a *prepare request* containing the value of *next* to some set *Proc* of processes (*Proc* represents a majority of the processes in the system).
2. Upon receipt of a *prepare request* containing a request number r from a process i, process j compares the received r with the value p that it has stored in its prepare request number field. If $p < r$, process j sets its prepare request number field to r and sends back to i a *prepare request* message containing its current belief.
3. After receiving a response for its *prepare request* message from a majority of processes and for a request number equal to the number stored in the corresponding field, process i selects, among all the received beliefs, the one with the most recent stamp. It then sends such belief, together with the current request number, to all the processes in *Proc* through an *accept request*.
4. Upon receipt of an *accept request* for a request number equal to its prepare request number field, process j changes its old belief with the one included in the received message and replies to the *accept request*.
5. If process i receives a response to its *accept request* from a majority of processes and for a request number equal to the number stored in the corresponding field, it then writes into its decision field the belief if proposed in the *accept request* and sends a *success* message containing such belief to every process in the system.
6. Upon receiving a *success* message, process j enters the belief con-

tained in such message into its decision field.

As Lamport says in [35], we can imagine that there are three roles in the Paxos algorithm and that these roles are performed by three classes of participants: proposers, acceptors and learners. The participants that belong to the proposers class are the ones that are allowed to start new rounds, collect estimates and choose values. The participants that belong to the acceptors class are the ones that simply wait for the proposers requests and that communicate their estimates. The participants that belong to the learners class are the ones that wait to know what the outcome value of the algorithm is. Of course, each process in the algorithm does belong to all the three classes and can be, at the same time, a proposer for request number $\hat{r}$, an acceptor for proposal request $\tilde{r}$, and a learner waiting to know about an outcome. This is the reason why each process has to permanently store the values in the record: *request number* is used by the process when it plays proposer, *prepare request number* and *belief* are used by the process when it plays acceptor, and *decision* is used by the process when it plays learner.

Processes can crash and recover at any time. When a process crashes, all messages that it had previously received are lost, as well as all messages that it was about to send. When it recovers, the only information that it retrieves from the past is its four field record.

3.2.2 Underlying Abstractions

Even though they are defined less clearly than in CT, also the Paxos algorithm needs the presence of some underlying abstractions. Starting from the specifications that we can find in the algorithm description [35, 36], we can fix the properties that each one of the abstractions should have, and in this way we can split the algorithm from the lower level modules.

The algorithm assumes that the processes are fully interconnected through bidirectional, duplicating and lossy communication channels.

There is a mechanism for electing leaders: each process is coupled with a module that is in contact with the modules of all the other processes. A process can query its module and learn whether it is leader or not. However, due to the asynchrony of the network, the modules are unreliable. So it can happen that, at the same point in time, there is more than one leader in the system, or maybe none at all. We consider the modules as being parts of a unique abstraction that we call *leader election*. Such abstraction can be identified with the Ω failure detector: eventually both of them output a process that is not crashed and has to be trusted.

When a process succeeds, it is mandatory that all the other correct processes receive the same *success* message. Since the channels are not reliable, it becomes necessary to use a communication protocol that gives more guarantees than the simple message sending through the network. As for the Chandra and Toueg algorithm, we make use of Reliable Broadcast. This protocol allows to send a message to all the processes in the system, assuring that (at least) all the correct processes will eventually receive the message. It guarantees three properties: if a correct process broadcasts a message, it will also eventually deliver it; if a correct process delivers a message, then all correct processes eventually also deliver the message; a process can deliver a message at most once, and only if the message had previously been broadcasted by a process.

3.2.3 An Intuitive Argument of Correctness

Intuitively the algorithm works because, like in CT, leaders always wait for a majority of messages before choosing a value to be sent with *accept request* and before taking a decision and spreading it with a *success* message.

Validity holds because none of the clauses of the algorithm invents any value. Whenever a value is written, it is always *copied* from some other data structure, or selected among several such sources. Obviously then, all values that occur in a reachable system configuration must originate from some initially given value.

As for the Chandra and Toueg algorithm, Agreement can intuitively be explained by means of a three-epoch structure of runs. In epoch 1 everything is possible, i.e., every initial value might eventually be decided. Moving to epoch 2 means that a value *gets locked*, i.e., for some round r (actually: the smallest such) a majority of processes send a positive acknowledgment to the leader owning round number r. Since the leader uses a maximal time-stamp strategy when choosing its proposal, and since any meaningful step forward in the algorithm requires a majority of processes to participate, a locked value v will always dominate the other values: should some value get locked in a greater round, then it will be the very same value v that was previously locked. In epoch 3 all (correct) processes decide the locked value. Due to the Reliable Broadcast properties, if a correct process decides, then all other correct processes decide as well.

Chapter 4

How Do We Formalize?

In the previous chapters we discussed the lack of formality in the description of algorithms when using natural language or pseudo code. In order to counter this incompleteness, we need to develop appropriate description techniques that incorporate the lacking information. Moreover, to counter the observed ambiguity in the semantics of the natural language or of the pseudo code with respect to the underlying system model, we propose to build the algorithm upon a formal description.

4.1 Our Approach

Our idea is to propose a way of writing distributed algorithms that is simple enough to be quickly available to anybody, but at the same time detailed and formal enough not to let any place to interpretation and misunderstanding. We express algorithms with the help of inference rules, obtaining algorithm descriptions that resemble rewrite systems. Thus, when applying transition rules to a system, we create an (in)finite tree of *runs* that describe all the possible evolutions of the system. The advantages of this strategy are multiple. The first is that the user does not have to learn any new language: a simple, superficial knowledge of the way inference rules work is more than enough. The second is that every rule corresponds (more or less) to an action of the algorithm as the writer would have put it in pseudo-code: a rule can be executed only if the premises are all verified and the execution implies some changes in the system. The third is that the premises of a rule are specified in details and the consequences of the execution are clearly stated for all the components of the system, this prevents all the ambiguities introduced with the use of pseudo-code. Moreover, the representation of a whole algorithm using inference rules usually occupies roughly one page, thus avoiding the reader to browse around the document looking for actions and conditions to apply. We believe that the structure is highly re-usable: we spent a long time preparing and tuning the first algorithm, but writing a second one was a matter of few days.

4.1.1 Modules

In order to provide a complete account of the behavior of an algorithm, we need to identify all the underlying modules that are needed for the execution, plus their role in the description and the correctness proofs of the algorithm. Typically, modules store pieces of information related to the execution. Such information is used to influence the evolution of the run by preventing some actions from being taken or by enabling the execution of other actions. Moreover, the information can be used in the correctness proofs to retrace the evolution of the system. When a module has to provide some specific properties, we represent them by pruning from the tree of possible runs the ones where the properties are violated. The result of this operation is a tree where all the branches represent a run of the system that satisfies the module properties.

We give here some examples of modules, stating the kind of information that needs to be stored, its role in the execution and/or in the proofs and the effect of the module properties on the system runs.

In order to be able to provide the properties of Accuracy and Completeness, failure detectors need to store information about all crashed processes and about the processes that are considered by others as ”trusted” (that are never going to crash). Thus, if we want to describe how a system incorporating $\Diamond S$ failure detectors behaves, we have to prune from the tree all the runs that do not respect the Eventual Accuracy and Completeness properties, i.e.,all runs where there is not a process that is ”trusted” after a certain time and where correct processes keep on forever suspecting correct processes.

An unreliable channel communication module needs to keep track of the messages that have been sent, duplicated, lost and received throughout the whole execution of the algorithm. This information influences the execution in the sense that the sending of a message enables an action allowing to receive the message. However, since such module does not have any special property, it does not require us to prune the run tree.

A refinement of the previous example is quasi-reliable point to point communication. As for the unreliable channel communication module, we need to keep track of messages sent and received, and this information influences the possible executions of the system. However, since the module has some additional properties, in order to model the fact that the communication is quasi-reliable we also need to prune all runs where messages sent by a correct process are not received by a correct process.

Once we have identified the underlying modules that are needed for the algorithm to work correctly, we provide a mathematical structure that describes the global idealized run-time system comprising all processes and the modules. Modules also play the role of a system history that keep track of relevant information during computation.

4.1.2 State

To keep track of the processes' states and of their changes, we use a component S defined as an array of n elements where n is the number of processes in the system and $\mathbf{S}(i)$ represents the state of process i.

The state of each process is a tuple, i.e. $\mathbf{S}(i) = (s_1, s_2, \ldots, s_\mu)$, where each component represents a piece of data that is necessary to the process for the execution of the algorithm. The number of components and their content need thus to be tailored to the specific algorithm under formalization. Specific to the algorithm and the system is also the way of formalizing crashes. If a process can be affected only by crash failures (without recoveries) we can say that the state of a crashed process is completely empty: $\mathbf{S}(i) = \bot$. This is correct since, usually, crashed processes completely stop any activity and lose memory of the past, so that not even other processes can access pieces of information stored on the crashed site. If instead the system allows recoveries, usually processes need to remember some data from the past: most often their incarnation number ι, but sometimes (according to the algorithm) also values that they need to know if they want to be able to continue with the execution when recovering. In this case, we indicate the process being up ($\top$) or crashed ($\bot$) by using a dedicated tuple component and a crash of the process does not necessarily empty all other fields of the state tuple: $\mathbf{S}(i) = (s_1, s_2, \ldots, \bot/\top, \iota, \ldots, s_\mu)$.

4.1.3 Configurations and Runs

We define a *configuration* as the combination of all the modules that are present in the system and are used by the algorithm, plus the state of the n processes that are running the algorithm. Formally, an algorithm is defined in terms of transition rules, that define computation steps as transitions between configurations of the form:

$$\langle\, \mathbf{M}1, \mathbf{M}2, \ldots \mathbf{M}\Gamma, \mathbf{S} \,\rangle \;\rightarrow \langle\, \mathbf{M}1', \mathbf{M}2', \ldots \mathbf{M}\Gamma', \mathbf{S}' \,\rangle$$

or, in vertical notation:

$$\left\langle \begin{array}{c} \mathbf{M}1 \\ \mathbf{M}2 \\ \vdots \\ \mathbf{M}\Gamma \\ \mathbf{S} \end{array} \right\rangle \quad \rightarrow \quad \left\langle \begin{array}{c} \mathbf{M}1' \\ \mathbf{M}2' \\ \vdots \\ \mathbf{M}\Gamma' \\ \mathbf{S}' \end{array} \right\rangle$$

where $\mathbf{M}1$, $\mathbf{M}2$,$\cdots$ $\mathbf{M}\Gamma$ represent the modules that store pieces of information, while $\mathbf{S}$ models the *state* of the n processes.

The symbol $\rightarrow$ denotes an action that can be executed by the system if the activation hypothesis are verified and that is typically presented as

$$\text{(RULE)}\ \frac{\text{Conditions on some of the configuration components}}{\langle\, \mathbf{M}1, \mathbf{M}2, \ldots \mathbf{M}\Gamma, \mathbf{S} \,\rangle \;\rightarrow \langle\, \mathbf{M}1', \mathbf{M}2', \ldots \mathbf{M}\Gamma', \mathbf{S}' \,\rangle}$$

Obviously, the most important condition that should appear in every rule is that the process executing the rule is not crashed at the moment of execution.

The symbol $\rightarrow^{*}$ denotes the reflexive-transitive closure of $\rightarrow$ (usually formally defined in a table containing all the possible actions of the system). The symbol $\rightarrow^{\infty}$ denotes an infinite sequence of $\rightarrow$. A *run* of an algorithm is then represented by sequences

$$\langle\, \mathbf{M}1_0, \mathbf{M}2_0, \ldots \mathbf{M}\Gamma_0, \mathbf{S}_0 \,\rangle \;\rightarrow^{*} \langle\, \mathbf{M}1, \mathbf{M}2, \ldots \mathbf{M}\Gamma, \mathbf{S} \,\rangle \;\rightarrow^{\imath} \qquad \text{where } \imath \in \{\, *, \infty \,\}$$

where $\langle\, \mathbf{M}1_0, \mathbf{M}2_0, \ldots \mathbf{M}\Gamma_0, \mathbf{S}_0 \,\rangle$ denotes the initial configuration of the modules and the processes in the system. We require runs to be complete, i.e., either infinite, or finite but such that they cannot be extended by some computation step in their final configuration.

Notation Most derivations of computation steps

$$\langle\, \mathbf{M}1, \mathbf{M}2, \ldots \mathbf{M}\Gamma, \mathbf{S} \,\rangle \;\rightarrow \langle\, \mathbf{M}1', \mathbf{M}2', \ldots \mathbf{M}\Gamma', \mathbf{S}' \,\rangle$$

are fully determined by (1) the source configuration $\langle\, \mathbf{M}1, \mathbf{M}2, \ldots \mathbf{M}\Gamma, \mathbf{S} \,\rangle$, (2) the identifier i of the process that executes the action —when each rule "touches" at most one state component— and (3) the name (RULE) of the rule that is applied. Accordingly, in statements about runs, we will occasionally denote computation steps by writing

$$\langle\, \mathbf{M}1, \mathbf{M}2, \ldots \mathbf{M}\Gamma, \mathbf{S} \,\rangle \;\rightarrow_{i:(\textsc{Rule})} \langle\, \mathbf{M}1', \mathbf{M}2', \ldots \mathbf{M}\Gamma', \mathbf{S}' \,\rangle \;.$$

Sometimes, instead of the former notation for runs,

$$\langle\, \mathbf{M}1_0, \mathbf{M}2_0, \ldots \mathbf{M}\Gamma_0, \mathbf{S}_0 \,\rangle \;\rightarrow^{*} \langle\, \mathbf{M}1, \mathbf{M}2, \ldots \mathbf{M}\Gamma, \mathbf{S} \,\rangle \;\rightarrow^{*/\infty}$$

we use the more succinct notation $\big(\mathfrak{C}_x\big)_T^{(v_1,\ldots,v_n)}$ as an abbreviation for

$$\big(\mathfrak{C}_x\big)_{x\in T}^{(v_1,\ldots,v_n)} \qquad \text{where} \qquad \mathfrak{C}_x = \langle\, \mathbf{M}1_x, \mathbf{M}2_x, \ldots \mathbf{M}\Gamma_x, \mathbf{S}_x \,\rangle$$

and T is an interval of natural numbers, i.e., either $[\,t_0, t_m\,]$ (denoting finite runs, or prefixes thereof) or $[\,t_0, \infty\,)$ (denoting infinite runs) for some $t_0, t_m \in \mathbb{N}$ with $t_0 \leq t_m$, such that for all $t > t_0$:

$$\mathfrak{C}_{t-1} \;\rightarrow_{i:(\textsc{Rule})} \mathfrak{C}_t$$

for some $i \in \mathbb{P}$ and rule (RULE). Unless specified otherwise, we assume $t_0 = 0$.

If we know that there is an appropriate instance of an expression, but its actual value is not important in the context, we use a dot $\cdot$ as a wildcard in order to ease the understanding of the important traits of the discussion. So if, for example, we are simply interested in the values of module $\mathbf{M}1$ and of $\mathbf{S}$, but the values of the other modules are not important for us at this point, we write:

$$\langle \mathbf{M}1, \cdot, \ldots \cdot, \mathbf{S} \rangle$$

Correct Processes The correctness of a process is always defined with respect to a specific run and each algorithm has specific requirements for defining a process as being correct in its runs. We generally indicate with $\mathsf{Correct}(R)$ the set of processes that can be defined as correct in run R of an algorithm. When needed (see Chapter 6), for some specific algorithm and some specific system, we will formally give a definition of correct processes.

Chapter 5

Formalization of Underlying Modules

As we have seen in Chapter 3, both the Chandra and Toueg and the Paxos Consensus algorithms work in systems with three underlying modules: a (more or less reliable) network, a Reliable Broadcast abstraction and a Failure Detection/Leader Election mechanism. As described in Chapter 4, the network, the Reliable Broadcast abstraction and the Failure Detection/Leader Election mechanism can all be represented through modules. Thus, the configurations describing the two Consensus algorithms are made of four components: the network module, the Reliable Broadcast module, the Failure Detection/Leader Election module and the general state of the processes.

In this Chapter, we present the three underlying modules, showing that they are independent one from the others and trying to give a description that is not strictly tailored to a particular algorithm, thus making the description reusable. In order to simplify the algorithm descriptions, we also decide to make a particular choice. Since the Ω Failure Detector/Eventual Leader Election module requires us to store only one piece of information (the unique eventual trusted process), we choose to describe Ω Failure Detection/Eventual Leader Election mechanism by simply imposing conditions on run executions.

5.1 The Network

The component $\mathbf{C}$ is a *global* data structure that contains CH-messages and their transmission status. The idea is that a particular $\mathbf{C}$ is associated to a particular instant in time referring to a particular run. More precisely, an instantaneous **C** provides access to all the CH-messages that have been sent (and possibly received, duplicated or lost) by the n processes up to that instant. This representation is convenient for both the formal description of the algorithm and its formal verification.

In order to keep track of all CH-messages ever sent during a run of the algorithm, we use a (multi-)set that is suitably defined as follows: for every execution of an instruction

$$\text{SEND}(j, r, \textit{contents}) \text{ TO } k$$

we record the information as a tuple of the form

$$(r, j, k, \textit{contents}) \ .$$

This has to be read as "the message with content *contents* that is sent in round r by process j to process k".

However, this is not enough to model the lifetime of CH-messages since generic network channels, a part from sending and delivering, can also duplicate and lose messages. Moreover, since the sending of multiple messages is usually not considered as an atomic action and if a process crashes while sending the same message to many different processes some processes might receive the message while others do not, we can say that messages can go through different transmission states:

outgoing**:** messages that, by the execution of some SEND statement, are put into an unordered buffer for *outgoing* messages at the sender site; the intuition is that such messages have not yet left the sender site, so a sender crash might still make them unavailable forever;

transit**:** messages that have left the sender site, but that have not yet arrived at their destination; these messages are kept in the unordered buffer of the transmission medium and are not affected in case of a sender crash, but they can instead be replicated or lost;

lost**:** messages that are stored within an unordered buffer for *lost* messages in a sort of "black hole" in the transmission medium;

received**:** messages that are stored—corresponding to the execution of some RECEIVE statement—within an unordered buffer for *received* messages at the intended receiver site; a message can be received more than once if it has been replicated during its transit in the transmission medium.

Figure 5.1 illustrates all the possible lifetimes of messages:

- When process i wants to send a message, it puts it into its Outgoing Buffer. Since the Outgoing Buffer is located inside Process i, if i crashes then all messages in the buffer get lost.

- From the Outgoing Buffer, the message can pass into the Network at any time. At this point, the message is subject only to the behavior of the transmission medium, and nothing that happens to Process i can any longer influence the message's course.
- While being in the Network, the message can be duplicated (even repeatedly), or can fall in the Lost Buffer, or both (some copies are lost, others are not).
- If not all the copies of the message are lost, the message can pass into the Received Buffer of the destination process j. Since the Received Buffer is located inside Process j, a crash of j destroys also all the messages contained in the buffer.

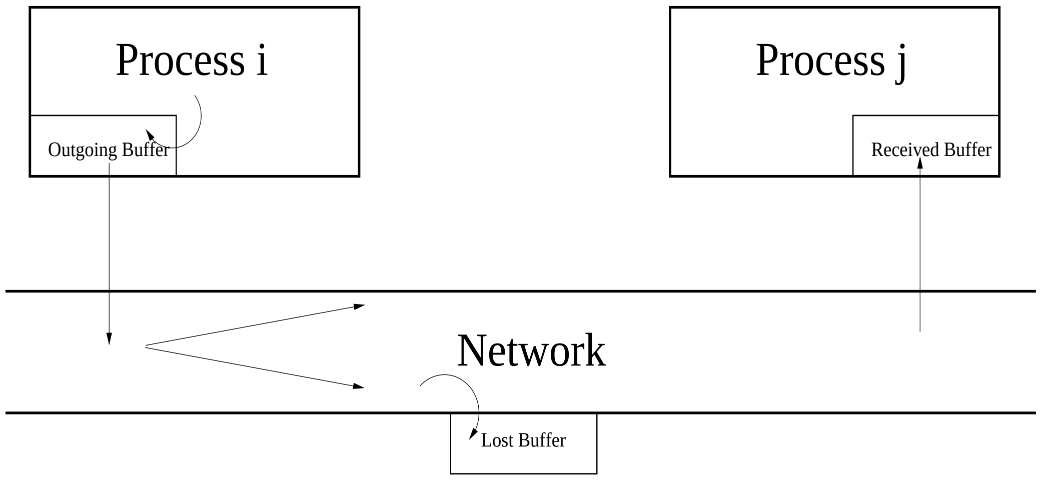

Figure 5.1: Possible lifetime of a message.

To model these transmission states in our data structure $\mathbf{C}$, we use a four component vector $(\alpha, \beta, \gamma, \delta)$ where $\alpha, \beta, \gamma, \delta \in \mathbb{N}$ and:

α indicates whether the message has left the outgoing buffer of the sender process or not. In particular, $\alpha = 0$ indicates that the message has been sent, while $\alpha \neq 0$ indicates that the message is in the outgoing buffer and it is ready to be sent. Note that since α represents a sort of yes/no flag, the value it assumes when it is not 0 is not really relevant. What counts is the distinction $\alpha = 0$ versus $\alpha \neq 0$.

β indicates that there are β copies of the message in transit in the network.

γ indicates that γ copies of the message have (already) been lost.

δ indicates whether at least one copy of the message has (already) been received or not. In particular, $\delta = 0$ indicates that the message has not yet been received, while $\delta \neq 0$ indicates that the message is available in the received buffer of the process and it is ready to be read.

This requires us to use (instead of the version above) extended tuples of the form

$$(r, j, k, \mathit{contents}, \tau)$$

where $\tau = (\alpha, \beta, \gamma, \delta)$. Note that, since it contains CH-messages in all transmission states, component $\mathbf{C}$ is not associated to a single location in the system.

We are now going to see how we can represent the behavior of the network in our formalization. Since, for the moment, we are not interested in expressing any algorithm, we simplify the study of the network by supposing that each process can send (i.e., put in its outgoing queue) the same message only once, and that the round number r, the sender j and the receiver k are enough to uniquely identify messages. As we will see in Chapters 8 and 9, these hypotheses are true for the algorithms under exam. For the same reason, we also suppose that the *contents* field of each message is empty. As the reader will notice, the presence or absence of contents in a message does not modify its transmission behavior, since the contents are used only to allow or disallow actions of the algorithm.

Taking advantage of the uniqueness of messages, in order to conveniently manipulate the data structure $\mathbf{C}$, we introduce the notation of "structured" *selectors* (representing the message's control information) of the form

$$P^{r}_{j\to k}.$$

where P is the message name, r is the round number in which the message has been sent, j is the sending process and k is the receiving process. Using structured selectors, we extract data from entries in $\mathbf{C}$ using the notation:

$$\mathbf{C}.P^{r}_{j\to k} \overset{\text{def}}{=} \begin{cases} \tau & \text{if } (r, j, k, \tau) \in \mathbf{C} \\ \perp & \text{otherwise} \end{cases}$$

We insert/update entries in $\mathbf{C}$ by the notation $\mathbf{C}\{\, X \mapsto V \,\}$ for $X = P^{r}_{j\to k}$ and $V \in (\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N})$ defined by:

$$\mathbf{C}\{\, X \mapsto V \,\}.P^{r}_{j\to k} \overset{\text{def}}{=} \begin{cases} V & \text{if } X = P^{r}_{j\to k} \\ \mathbf{C}.P^{r}_{j\to k} & \text{otherwise} \end{cases}$$

Using this notation, we can write the rules that model the lifetime of messages across transmission states. These rules are independent of the current value of component $\mathbf{S}$, and $\mathbf{S}$ is only used to ensure that the process involved in the sending is not crashed. This reflects the fact that the underlying transmission medium ships messages from node to node, duplicates and loses them independently of any algorithm running at the nodes of the system.

We suppose at first that processes in the system can crash but not recovery, and we propose a first formalization for this case. We will then see what we need to change in our representation if we want our model to be used in a crash-recovery failure model system.

5.1.1 Network in Crash Failure Model Systems

We assume that, when it is created, a message from j to k in round r has the form

$$\mathbf{C}.P^r_{j\to k} \overset{\text{def}}{=} (1,0,0,0)$$

Then, we have the following rules:

$$(\textsc{Ch\_snd})\ \frac{\mathbf{S}(j)\neq\bot \qquad \mathbf{C}.P^r_{j\to k}=(1,0,0,0)}{\langle\mathbf{C},\mathbf{S}\rangle\ \to\langle\mathbf{C}\{\,P^r_{j\to k}\mapsto(0,1,0,0)\,\},\mathbf{S}\rangle}$$

$$(\textsc{Ch\_deliver})\ \frac{\mathbf{S}(k)\neq\bot \qquad \mathbf{C}.P^r_{j\to k}=(\alpha,\mathrm{succ}(\beta),\gamma,\delta)}{\langle\mathbf{C},\mathbf{S}\rangle\ \to\langle\mathbf{C}\{\,P^r_{j\to k}\mapsto(\alpha,\beta,\gamma,\mathrm{succ}(\delta))\,\},\mathbf{S}\rangle}$$

$$(\textsc{Ch\_dupl})\ \frac{\mathbf{C}.P^r_{j\to k}=(\alpha,\beta,\gamma,\delta) \qquad \beta>0}{\langle\mathbf{C},\mathbf{S}\rangle\ \to\langle\mathbf{C}\{\,P^r_{j\to k}\mapsto(\alpha,\mathrm{succ}(\beta),\gamma,\delta)\,\},\mathbf{S}\rangle}$$

$$(\textsc{Ch\_loss})\ \frac{\mathbf{C}.P^r_{j\to k}=(\alpha,\mathrm{succ}(\beta),\gamma,\delta)}{\langle\mathbf{C},\mathbf{S}\rangle\ \to\langle\mathbf{C}\{\,P^r_{j\to k}\mapsto(\alpha,\beta,\mathrm{succ}(\gamma),\delta)\,\},\mathbf{S}\rangle}$$

If channels cannot replicate or lose messages , rules (CH_DUPL) and (CH_LOSS), as well as γ, are not necessary to express the behavior of the network. We then have simply:

$$(\textsc{Ch\_snd})\ \frac{\mathbf{S}(j) \neq \bot \qquad \mathbf{C}.P^r_{j\to k} = (1,0,0)}{\langle \mathbf{C}, \mathbf{S}\rangle \ \rightarrow \langle \mathbf{C}\{\, P^r_{j\to k} \mapsto (0,1,0)\,\}, \mathbf{S}\rangle}$$

$$(\textsc{Ch\_deliver})\ \frac{\mathbf{S}(k) \neq \bot \qquad \mathbf{C}.P^r_{j\to k} = (0,1,0)}{\langle \mathbf{C}, \mathbf{S}\rangle \ \rightarrow \langle \mathbf{C}\{\, P^r_{j\to k} \mapsto (0,0,1)\,\}, \mathbf{S}\rangle}$$

Looking at these two rules, we notice that in the case of Crash Failure Model Systems:

- α, β and δ are equal to 1 in a mutually exclusive way,
- α, β and δ never take a value that is greater than 1,
- once α and β have passed from value 1 to value 0 and δ has passed from value 0 to value 1 nothing more can happen.

Starting from such considerations, we can drop the representation of the tag with three components and adopt a more compact representation of a tag with only one component that will be able to take three different values called $\mathrm{outgoing}$, $\mathrm{transit}$ and $\mathrm{received}$. The rules then become:

$$(\textsc{Ch\_snd})\ \frac{\mathbf{S}(j) \neq \bot \qquad \mathbf{C}.P^r_{j\to k} = (\mathrm{outgoing})}{\langle \mathbf{C}, \mathbf{S}\rangle \ \rightarrow \langle \mathbf{C}\{\, P^r_{j\to k} \mapsto (\mathrm{transit})\,\}, \mathbf{S}\rangle}$$

$$(\textsc{Ch\_deliver})\ \frac{\mathbf{S}(k) \neq \bot \qquad \mathbf{C}.P^r_{j\to k} = (\mathrm{transit})}{\langle \mathbf{C}, \mathbf{S}\rangle \ \rightarrow \langle \mathbf{C}\{\, P^r_{j\to k} \mapsto (\mathrm{received})\,\}, \mathbf{S}\rangle}$$

5.1.2 Network in Crash-Recovery Failure Model Systems

If the system supports a crash-recovery failure model, then the message tag needs to keep track of the incarnation number of the sending and the receiving processes. In particular, since a process that has crashed and then recovered in incarnation ι does not have the right to send messages that were put into the outgoing buffer by an incarnation preceding ι (at each crash all the outgoing messages are lost), each outgoing message must carry the incarnation value of the sender and rule (CH_SND) must be enabled only if the incarnation value stored in the message equals the current incarnation value of the process. The same holds for the receiver: since a process is entitled to use only the messages that were received since its last recovery (or since the beginning of the algorithm, if it never crashed), each received message must carry the incarnation value of the receiving process at the time of reception, and any rule of the algorithm that sits on top of the network and uses received messages has to check if the current incarnation of the process matches the incarnation that is stored in the message.

In theory, these two constraints would force us to introduce two components more in the message tag. However, we can avoid this by using the α and the δ components in a better way. In fact, we can see that α works like a flag (it simply signals that a message is ready to be sent through the network) and its value, when different from 0, is not important. Moreover, when the message is sent through the network, α is set again to 0 and never changes its value again. So, we can use α to carry the incarnation number of the sending process: when willing to send a message, a process copies its incarnation number into α instead of simply setting it to 1. If we suppose that the state component of each process stores only the status (up or down) and the incarnation number of the process, rule (CH_SND) becomes as follows:

$$\text{(CH\_SND)}\ \frac{\mathbf{S}(j) = (\top, \iota) \qquad \mathbf{C}.P^r_{j\to k} = (\iota, 0, 0, 0)}{\langle \mathbf{C}, \mathbf{S}\rangle \ \rightarrow \langle \mathbf{C}\{\, P^r_{j\to k} \mapsto (\iota, 1, 0, 0)\,\}, \mathbf{S}\rangle}$$

Notice that rule (CH_SND) is enabled only when α is different from 0 and β, γ and δ are all equal to 0. Once a message is put into the network, at least one among β, γ and δ is always greater than 0, so the fact that we do not reset α to 0 after sending does not present a problem from the point of view of multiple message sending (after the first execution, rule (CH_SND) is no longer enabled for the same message).

Since the replication and the loss of a message does not depend on the incarnation number of any process, rules (CH_DUPL) and (CH_LOSS) are the same as in Section 5.1.1.

Since a process is entitled to access only the messages that were received during its current incarnation, the incarnation number has to be present in the message tag. To figure out how we can do this, we can look at δ. δ signals that a message has been received and it increases its value at every reception of a message copy. Since usually, in the algorithms that we study, it is irrelevant how many copies of the message have been received, provided that at least one has, we can say that our rule (CH_DELIVER) is enabled only if a message has not yet been received. Since, if a process crashes and recovers in a new incarnation, its incoming message buffer is empty, the same message can be received again if there is still a copy in transit and if the incarnation number of the process is different. So, we use δ to store the incarnation number of the receiving process, and we allow a new reception only if the value of δ is strictly smaller than the current incarnation number of the receiving process. Thus, rule (CH_DELIVER) looks as follows:

$$(\text{CH\_DELIVER})\ \frac{\mathbf{S}(k) = (\top, \iota) \qquad \mathbf{C}.P^{r}_{j\to k} = (\alpha, \operatorname{succ}(\beta), \gamma, \delta) \qquad \delta < \iota}{\langle \mathbf{C}, \mathbf{S}\rangle \ \to \langle \mathbf{C}\{\, P^{r}_{j\to k} \mapsto (\alpha, \beta, \gamma, \iota)\,\}, \mathbf{S}\rangle}$$

5.1.3 Network Properties

Very often, distributed algorithms have the desired behavior only if the network provides some specific properties in the run. To model this constraint, we first formally describe the required properties, then we prune, from the list of all possible runs, all runs that do not respect the properties.

To describe a property, we take the network representation that suits the system in which our algorithm runs, then we impose conditions on the actions that should ”eventually” take place in the run, if other actions were previously executed. So, if we want to use our formalism to express the quasi-reliability property required by the Chandra Toueg algorithm for non lossy and non replicating channels, we have:

Definition 5.1.3.1 (Quasi-Reliable Point-To-Point) *Let* $R = \left(\text{CH}_x, \mathbf{M}2_x, \ldots \mathbf{M}\Gamma_x, \mathbf{S}_x\right)_T$ *denote a run.* R *satisfies* QPP-Reliability *if, for all processes* $j, k \in \mathsf{Correct}(R)$ *and time* $t > 0$*, whenever*

$$\langle \text{CH}_{t-1}, \cdot, \ldots \cdot, \cdot\rangle \ \to_{j:(\cdot)} \langle \text{CH}_{t}, \cdot, \ldots \cdot, \cdot\rangle$$

with $\text{CH}_{t-1}.P^{r}_{j\to k} = \bot$ *and* $\text{CH}_{t}.P^{r}_{j\to k} = (\text{outgoing})$*, for* $r > 0$*, then there is* $u > t$ *such that*

$$\langle \text{CH}_{u-1}, \cdot, \ldots \cdot, \cdot\rangle \ \to_{k:(\text{CH\_DELIVER})} \langle \text{CH}_{u}, \cdot, \ldots \cdot, \cdot\rangle$$

with $\text{CH}_{u}.P^{r}_{j\to k} = (\text{received})$.

5.2 Reliable Broadcast

Reliable Broadcast is an abstraction that allows a process to reliably send a message to all the other (non-crashed) processes in the system. If we call RBC_BROADCAST$(\mathit{sender}, \mathit{contents})$ the action of ”reliably sending” a message and RBC_DELIVER(s, c) the action of ”reliably delivering” a message, Reliable Broadcast guarantees the following three properties:

Validity

If a correct process executes RBC_BROADCAST$(\mathit{sender}, \mathit{contents})$, then it also eventually executes RBC_DELIVER(s, c) for the message $(\mathit{sender}, \mathit{contents})$.

Agreement

If a correct process executes RBC_DELIVER(s, c) for some message $(\mathit{sender}, \mathit{contents})$, then all correct processes eventually execute RBC_DELIVER(s, c) for this message $(\mathit{sender}, \mathit{contents})$.

Uniform Integrity

For any message $(\mathit{sender}, \mathit{contents})$, every process executes RBC_DELIVER(s, c) at most once, and only if some process previously executed RBC_BROADCAST$(\mathit{sender}, \mathit{contents})$.

Since Reliable Broadcast gives many more guarantees (as for message delivery) than simple message passing over the network, its usage is far more complex–and far more expensive–than the simple use of the network. Thus, distributed algorithms usually reliably broadcast only crucial messages. In particular, most consensus algorithms use Reliable Broadcast exclusively for disseminating the final decision: in a given round, only the process that is leading the round can, if it is able to gather enough acknowledgments for a proposed value, execute a RBC_BROADCAST action to trigger the sending of messages that contain the value.

In order to satisfy the RBC-properties, our model must keep track of the delivery status of each broadcast message with respect to every involved process. In particular, for each broadcast message, we have to remember which processes have already delivered the message and which processes have not yet done it. A run of a system that includes a Reliable Broadcast module is then defined acceptable either if the sender of the message is not correct and none of the correct processes in the system ever delivers the message, or if none of the processes that never deliver the message is correct.

In order to keep track of what messages are broadcast and of what happens to them during a run, we can introduce a data structure similar to what we have introduced for describing the network. We call this new component B (for Reliable *B*roadcast) and we structure it as a (multi-)set in which each element contains (i) the broadcast message and (ii) the set of processes that are supposed to deliver it. Since, as we have said, only the round leader is usually allowed to broadcast a message, and since for every round there is only one possible leader, it makes sense to use round numbers to properly distinguish all the broadcasts that may occur in a run. Thus, we model B as an unbounded array indexed by the round number at the moment of sending. So, at any time of a run and for any round number $r \in \mathbb{N}$, $\mathbf{B}(r)$ is either undefined ($\perp$) or it is a pair of the form (v, D) where $v \in \mathbb{V}$ (set of possible values) is the broadcasted value, and $D \subseteq \mathbb{P}$ (set of processes in the system) is the set of processes that have not yet delivered this message. Initially $\mathbf{B}_0 := \emptyset$, that is $\mathbf{B}_0(r) = \perp$ for all $r \in \mathbb{N}$. We insert/update entries in $\mathbf{B}$ by the notation $\mathbf{B}\{\, r \mapsto V \,\}$ for $r \in \mathbb{N}$ and $V \in (\mathbb{V} \times 2^{\mathbb{P}})$ defined by:

$$\mathbf{B}\{\, r \mapsto V \,\}(\hat{r}) \overset{\text{def}}{=} \begin{cases} V & \text{if } \hat{r} = r \\ \mathbf{B}(\hat{r}) & \text{otherwise} \end{cases}$$

Supposing that the only configuration components present in the system are the process states and Reliable Broadcast, the rules that express the sending and receiving messages for Reliable Broadcast can be written as follows:

$$(\textsc{Rbc-broadcast})\ \frac{\text{Some conditions}}{\left\langle \begin{array}{c} \mathbf{B} \\ \mathbf{S} \end{array} \right\rangle \rightarrow \left\langle \begin{array}{llll} \mathbf{B}\{ & r & \mapsto & (v, \mathbb{P}) \quad \} \\ \mathbf{S}\{ & i & \mapsto & \text{Following state } \} \end{array} \right\rangle}$$

$$(\textsc{Rbc-deliver})\ \frac{\begin{array}{c} \mathbf{S}(i) \neq \perp \quad \text{or} \quad \mathbf{S}(i) = (\top, \iota) \\ \mathbf{B}(r) = (v, D) \qquad i \in D \end{array}}{\left\langle \begin{array}{c} \mathbf{B} \\ \mathbf{S} \end{array} \right\rangle \rightarrow \left\langle \begin{array}{llll} \mathbf{B}\{ & r & \mapsto & (v, D \backslash \{i\}) \quad \} \\ \mathbf{S}\{ & i & \mapsto & \text{Changes implied by delivery } \} \end{array} \right\rangle}$$

Where ”Some conditions” are the conditions imposed by the algorithm to generate a broadcast, ”Following state” is the state that the broadcasting process i attains after having broadcasted a message, and ”Changes implied by delivery” are the changes to the state of process i (the one executing the delivery) that are produced by the delivery of the message.

As we can see, Reliable Broadcast does not change according to the properties of the system (reliable or unreliable communication, crash or crash-recovery failures). The only condition Reliable Broadcast imposes is related to the process that wants to execute the broadcast or the deliver–it is supposed to be alive at the moment of execution of the action. All the other characteristics of the system are masked by the Reliable Broadcast abstraction.

5.2.1 Reliable Broadcast Properties

In order to represent the properties that make Reliable Broadcast a very valuable module to use in a system, we need to define (as we have done for the Network) a set of conditions on runs that allow us to prune the possible run tree and make us consider only the runs that respect the desired properties. For the formal definition of the properties of Reliable Broadcast we have a direct correspondence between the informal terminology of "executing a primitive" and the execution of a particular transition rule.

Definition 5.2.1.1 (Reliable Broadcast) *Let* $R = \big(\mathbf{B}_x, \mathbf{M2}_x, \ldots \mathbf{M\Gamma}_x, \mathbf{S}_x\big)_T$ *denote a run. Then we define some of its properties:*

1. RBC-Validity: *for every* $i \in \mathsf{Correct}(R)$*, if there is* $t > 0$ *such that*

$$\langle \mathbf{B}_{t-1}, \cdot, \ldots \cdot, \cdot \rangle \rightarrow_{i:(\text{RBC-BROADCAST})} \langle \mathbf{B}_t, \cdot, \ldots \cdot, \cdot \rangle$$

with $\mathbf{B}_{t-1}(r) = \bot$ *and* $\mathbf{B}_t(r) = (v, \mathbb{P})$ *for some* $r > 0$*, then there is* $u > t$ *such that*

$$\langle \mathbf{B}_{u-1}, \cdot, \ldots \cdot, \cdot \rangle \rightarrow_{i:(\text{RBC-DELIVER})} \langle \mathbf{B}_u, \cdot, \ldots \cdot, \cdot \rangle$$

with $\mathbf{B}_{u-1}(r) = (v, D_{u-1})$*,* $\mathbf{B}_u(r) = (v, D_u)$ *and* $D_{u-1} = D_u \uplus \{i\}$*.*

2. RBC-Agreement: *for every* $i \in \mathsf{Correct}(R)$*, if there is* $t > 0$ *such that*

$$\langle \mathbf{B}_{t-1}, \cdot, \ldots \cdot, \cdot \rangle \rightarrow_{i:(\text{RBC-DELIVER})} \langle \mathbf{B}_t, \cdot, \ldots \cdot, \cdot \rangle$$

with $\mathbf{B}_{t-1}(r) = (v, D_{t-1})$*,* $\mathbf{B}_t(r) = (v, D_t)$ *and* $D_{t-1} = D_t \uplus \{i\}$ *for some* $r > 0$*,*
then, for every $j \in \mathsf{Correct}(R)$*, there is* $u > 0$ *such that*

$$\langle \mathbf{B}_{u-1}, \cdot, \ldots \cdot, \cdot \rangle \rightarrow_{j:(\text{RBC-DELIVER})} \langle \mathbf{B}_u, \cdot, \ldots \cdot, \cdot \rangle$$

with $\mathbf{B}_{u-1}(r) = (v, D_{u-1})$*,* $\mathbf{B}_u(r) = (v, D_u)$ *and* $D_{u-1} = D_u \uplus \{j\}$*.*

3. (Uniform) RBC-Integrity: *for every* $i \in \mathsf{Correct}(R)$*, if there are* $t_1, t_2 > 0$ *such that*

$$\langle \mathbf{B}_{t_1-1}, \cdot, \ldots \cdot, \cdot \rangle \rightarrow_{i:(\text{RBC-DELIVER})} \langle \mathbf{B}_{t_1}, \cdot, \ldots \cdot, \cdot \rangle$$

$$\langle \mathbf{B}_{t_2-1}, \cdot, \ldots \cdot, \cdot \rangle \rightarrow_{i:(\text{RBC-DELIVER})} \langle \mathbf{B}_{t_2}, \cdot, \ldots \cdot, \cdot \rangle$$

and, for some $r > 0$*, both*

   (a) $\mathbf{B}_{t_1-1}(r) = (v, D_{t_1-1})$*,* $\mathbf{B}_{t_1}(r) = (v, D_{t_1})$*, and* $D_{t_1-1} = D_{t_1} \uplus \{i\}$*.*
   (b) $\mathbf{B}_{t_2-1}(r) = (v, D_{t_2-1})$*,* $\mathbf{B}_{t_2}(r) = (v, D_{t_2})$*, and* $D_{t_2-1} = D_{t_2} \uplus \{i\}$*.*

then $t_1 = t_2$ *and there is* $u < t_1$ *such that*

$$\langle \mathbf{B}_{u-1}, \cdot, \ldots \cdot, \cdot \rangle \rightarrow_{i:(\text{RBC-BROADCAST})} \langle \mathbf{B}_u, \cdot, \ldots \cdot, \cdot \rangle$$

with $\mathbf{B}_{u-1}(r) = \bot$ *and* $\mathbf{B}_u(r) = (v, \mathbb{P})$*.*

5.3 On Histories, Locality and Distribution.

In the context of algorithm verification, it is often useful to book-keep some execution information that is needed for the proofs. This information should not be used by the processes as knowledge (i.e., it should not affect the behavior of the algorithm under study), it should rather be used to simplify the job of the prover. *History variables* have been introduced for this purpose, as an augmentation of the state space that is not accessible to the processes. (See the *Related work* section in §12 for a comparison with related approaches and the most relevant references.)

In our work, we use the components $\mathbf{B}$ and $\mathbf{C}$ as communication media. For the proofs, not only we need to record the messages sent or received by correct processes, but we also must keep track of the messages that possibly got lost due network malfunction or to process crashes. In order to avoid the repetition of alike structures, we have chosen to enrich $\mathbf{B}$ and $\mathbf{C}$ with additional information. This information does not affect the behavior of the algorithm because processes neither need such additional information nor can they ever take advantage of it. In the following, we clarify this double role of $\mathbf{B}$ and $\mathbf{C}$.

In $\mathbf{B}$, RBC-messages appear as soon as they are RBC-broadcast, while on RBC-delivery the intended receiver checks for the presence of a relevant message in the medium. The knowledge that a RBC-delivery of the message by some particular process has occurred remains present in $\mathbf{B}$, always leaving behind an entry of the form $(v, \cdot)$ as witness, even if all processes have RBC-delivered. However, we formalize the algorithm in such a way that no process can ever see whether another process has already RBC-delivered, so this globally available knowledge does not influence the algorithm.

With $\mathbf{C}$ and its interpretation given in §5.1, the situation is slightly more complicated, because the component does not only play the global role of a communication medium, but also the role of message queues, namely

$$\{\, x \in \mathbf{C} \mid x = (\cdot, \cdot, j, \cdot, \cdot, (\alpha, \cdot, \cdot, \cdot)) \,\}$$

for outgoing CH-messages, local at the sender (here: for process j), and

$$\{\, x \in \mathbf{C} \mid x = (\cdot, \cdot, \cdot, k, \cdot, (\cdot, \cdot, \cdot, \delta)) \,\}$$

for received CH-messages, local at the receiver (here: for process k). The double use of $\mathbf{C}$ as communication medium and message queues provides us with convenient mathematical proof structures. For example, outgoing message queues allow us to model the simultaneous sending of multiple messages by an atomic step of the algorithm. The crash-sensitive passage of individual messages from the outgoing queue to the communication medium is left to an independent rule, but still works on the same data structure by just changing message tags. As in the case of RBC-messages, CH-messages are never thrown away, but are rather kept in the data structure to witness their prior existence in the run.

Despite the apparently global availability of the structure $\mathbf{C}$, in particular of all of its queues, our formalization is built in such a way that the execution of any rule of the algorithm by any process maintains a purely local character. In fact, whenever some process checks for the presence of one or several messages in the medium, it always only checks for their presence in subcomponents that represent its own local message queues. Symmetrically, whenever some process sends a message, this is put in the process' own local outgoing queue.

Summing up, when considered as communication media both $\mathbf{B}$ and $\mathbf{C}$ model the complete lifetime of the respective kinds of messages. We consider a CH-message as *dead* in three different cases: (i) when it is stuck in the $\mathrm{outgoing}$-queue of a crashed process, (ii) when it is stuck in the $\mathrm{received}$-queue of a crashed process, (iii) when it is kept in a $\mathrm{received}$-queue and it has already been processed, (iv) when it is in $\mathrm{transit}$ and its receiver has crashed, and for lossy channels, (v) when it is lost and no other copy is in transit ($\beta = 0$).

We consider a RBC-message as *dead* if it is kept in $\mathbf{B}$ but the set of processes that still have to RBC-deliver it is either empty or contains only crashed processes. The history character of $\mathbf{B}$ and $\mathbf{C}$ is represented precisely by the fact that they keep information about each message even beyond the death of the message itself, which is in contrast to the mere algorithmic use. This kind of historical information would not be as easily accessible, and thus not as easy to model, if we did not integrate the multiple local CH-messages queues within the single global component $\mathbf{C}$.

5.4 Failure Detectors and Leader Election

To gain freedom from low implementation details, failure detectors and eventual leader election can be considered as idealized abstractions that govern the allowed mutual suspicion of processes about whether processes have crashed or not. In §2.2.1, we informally introduced the notion of the FD Ω (equivalent to the election of a leader), which is defined to satisfy the following property:

there is a time $\hat{t}$ after which all processes always trust (i.e., do not suspect) the same correct process k.

We see two slightly different ways in which we can model such property. One way consists in explicitly representing the set of crashed and the set of trusted processes, and in giving explicit rules through which a process can become part of one of these two sets. In this formalization, an action can be executed only by a process that is not yet part of the crashed process set. Moreover, each rule that involves a suspicion is conditioned by the fact that the suspected process should not be in the set of trusted processes. The second way of modeling the property is to keep the trusted processes implicit and not to add any rule to signal that a process has become trusted. Then, we simply identify the actions of the algorithm that require a process suspecting another process and prune the tree of possible runs by imposing constraints on the execution of such actions. The two approaches differ in the sense that the first one explicitly models the trust in a particular process k via some (global) mathematical structure (as the run-specific time-dependent function $H_\Omega : \mathbb{P} \times T \rightarrow \mathbb{P}$ of [11]), while the second one abstracts from such fine-grained description and rather just observes that process k *is* trusted in a given run. In order to keep the algorithm descriptions as simple and slender as possible, we choose the second modeling strategy [1]. However, since the explicit modeling is used for precisely the purpose of detecting a trusted process, we can consider the two representations as being equivalent.

As we have mentioned above, the way we have chosen for describing the Ω property must be tailored to the algorithm, since it has to enforce that, after a certain time, a particular rule of the algorithm (related to the fact of one process being trusted or not) is never again applied if some conditions are satisfied. For example, if we call the suspicion rule (SUSPECT) and if we want to formalize the property in our rewrite system by expressing that, whenever after time $\hat{t}$ some process i suspects another process then this suspected process is different from the trusted process k, we can write what follows:

Definition 5.4.0.2 (Failure Detection) *Let $R = \left(\mathbf{M}1_x, \mathbf{M}2_x, \dots \mathbf{M}\Gamma_x, \mathbf{S}_x\right)_T$ be a run. R satisfies the property Ω if there is $k \in \mathsf{Correct}(R)$ and there is $\hat{t} > 0$ such that for all $t > \hat{t}$, whenever*

$$\langle \cdot, \cdot, \mathbf{S}_{t-1} \rangle \rightarrow_{i:(\text{SUSPECT})} \langle \cdot, \cdot, \mathbf{S}_t \rangle$$

then k is not the target of the suspicion of i.

[1]The interested reader will find a description of the first strategy in §12.1.1.

If instead we want to express that, in a system with explicit leader election, eventually only one process becomes (or stays) leader, we can impose the following condition on the run:

Definition 5.4.0.3 (Eventual Leader Election) *Let* $R = \left(\mathbf{M}1_x, \mathbf{M}2_x, \ldots \mathbf{M}\Gamma_x, \mathbf{S}_x\right)_T$ *be a run.* R *satisfies the property of Eventual Leader Election if*

there is a time $t \in T$ *and a process* $i \in \mathsf{Correct}(R)$ *such that if*

$$\langle \cdot, \cdot, \ldots \cdot, \mathbf{S}_{t-1} \rangle \ \rightarrow_{i:(\text{LEADER-YES})} \langle \cdot, \cdot, \ldots \cdot, \mathbf{S}_t \rangle$$

then there is no $t' \in T$ *with* $t' \geq t$ *such that*

$$\langle \cdot, \cdot, \ldots \cdot, \mathbf{S}_{t'-1} \rangle \ \rightarrow_{i:(\text{LEADER-NO})} \langle \cdot, \cdot, \ldots \cdot, \mathbf{S}_{t'} \rangle$$

and there is a time $u \in T$ *such that, for any process* $j \in \mathbb{P}$ *there is no* $u' \in T$*, with* $u' \geq u$ *such that*

$$\langle \cdot, \cdot, \ldots \cdot, \mathbf{S}_{u'-1} \rangle \ \rightarrow_{j:(\text{LEADER-YES})} \langle \cdot, \cdot, \ldots \cdot, \mathbf{S}_{u'} \rangle$$

and for all processes $j \in \mathbb{P}$*, with* $j \neq i$*, if there is a time* $v \in T$ *such that*

$$\langle \cdot, \cdot, \ldots \cdot, \mathbf{S}_{v-1} \rangle \ \rightarrow_{j:(\text{LEADER-YES})} \langle \cdot, \cdot, \ldots \cdot, \mathbf{S}_v \rangle$$

then there is a $v' \in T$ *with* $v' \geq v$ *such that*

$$\langle \cdot, \cdot, \ldots \cdot, \mathbf{S}_{v'-1} \rangle \ \rightarrow_{j:(\text{LEADER-NO})} \langle \cdot, \cdot, \ldots \cdot, \mathbf{S}_{v'} \rangle$$

Chapter 6

Formalization of the Chandra-Toueg Consensus Algorithm

In Chapters 4 and 5 we have described our way of representing algorithms and underlying abstractions through the use of transition rules. Now we are going to apply such method to the Chandra-Toueg Consensus algorithm. In order to provide a complete account of the behavior of the algorithm, we need to explicitly represent the communication network as well as all interactions with it. This translates into having configurations that express the processes'states plus the buffering of QPP- and RBC-messages.

Thus, we define the Chandra and Toueg Consensus algorithm in terms of transition rules as follows:

$$\langle\, \mathbf{B}, \mathbf{Q}, \mathbf{S} \,\rangle \;\rightarrow \langle\, \mathbf{B}', \mathbf{Q}', \mathbf{S}' \,\rangle \quad \text{or, in vertical notation:} \quad \left\langle \begin{array}{c} \mathbf{B} \\ \mathbf{Q} \\ \mathbf{S} \end{array} \right\rangle \rightarrow \left\langle \begin{array}{c} \mathbf{B}' \\ \mathbf{Q}' \\ \mathbf{S}' \end{array} \right\rangle$$

where $\mathbf{B}$ and $\mathbf{Q}$ contain the (histories of) messages sent during the execution of the algorithm through the Reliable *B*roadcast service and through the *Q*uasi-reliable Point-to-Point service, respectively, while $\mathbf{S}$ models the *state* of the n processes.

A *run* of the Consensus algorithm is then represented by sequences

$$\langle\, \mathbf{B}_0, \mathbf{Q}_0, \mathbf{S}_0^{(v_1 \ldots, v_n)} \,\rangle \;\rightarrow^* \langle\, \mathbf{B}, \mathbf{Q}, \mathbf{S} \,\rangle \;\rightarrow^\imath \qquad \text{where } \imath \in \{\, *, \infty \,\}$$

where $\langle\, \mathbf{B}_0, \mathbf{Q}_0, \mathbf{S}_0^{(v_1, \ldots, v_n)} \,\rangle$ denotes the initial configuration based on the values $v_1, \ldots, v_n$ initially proposed by the n processes in the system.

6.1 Configurations

Next, we explain in detail the three components of a *configuration* $\langle \mathbf{B}, \mathbf{Q}, \mathbf{S} \rangle$.

6.1.1 Process State

When representing the algorithm by a set of step-oriented transition rules, we inevitably lose the explicit sequencing of statements in the former pseudo-code description of the algorithm. To remedy this loss of information, we model explicitly to which subsequent phase the algorithm needs to jump after the application of any transition rule. Thus, we enhance the "program counter" of individual processes to include not only their round number r, but also the label indicating their current phase P. We further introduce an additional label W, which we use to model the "entry phase" of `while`-loops.

Component $\mathbf{S}$ is defined as an array of n elements, where $\mathbf{S}(i)$, for $1 \leq i \leq n$, represents the state of process i. The state of a process is a triple of the form $\big(c,\, b,\, d\big)$, where

- $c = (r, P)$ is the *program counter* (always defined), consisting of a round number $r \in \mathbb{N}$ and a phase label $P \in \{\mathsf{W}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{P4}\}$;
- $b = (v, s)$ is the *belief* (always defined), consisting of a value estimate $v \in \mathbb{V}$ and a stamp $s \in \mathbb{N}$, representing the round in which the belief in the value was adopted;
- $d = (v, s)$ is the *decision* (may be undefined, i.e., $d = \bot$), consisting of a value $v \in \mathbb{V}$ and a stamp $s \in \mathbb{N}$, representing the round in which value v has been broadcasted by $\mathrm{crd}(s)$.

Every process i starts at round 0 and in phase W, where it simply believes in its own value v_i tagged with stamp 0; the decision field is left undefined. Thus, for every process i, the initial state is:

$$\mathbf{S}_0^{(v_1,\ldots,v_n)}(i) = \big((0, \mathsf{W}), (v_i, 0), \bot\big)$$

We write $\mathbf{S}\{\, i \mapsto V \,\}$ for $i \in \mathbb{P}$ and
$V \in ((\mathbb{N} \times \{\mathsf{W}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{P4}\}) \times (\mathbb{V} \times \mathbb{N}) \times ((\mathbb{V} \times \mathbb{N}) \cup \{\bot\})) \cup \{\bot\}$ to insert/update entries in $\mathbf{S}$ as follows:

$$\mathbf{S}\{\, i \mapsto V \,\}(k) \overset{\text{def}}{=} \begin{cases} V & \text{if } k = i \\ \mathbf{S}(k) & \text{otherwise} \end{cases}$$

In Table 6.1, with rule (CRASH), we model the crash of a process i by setting the respective ith entry of $\mathbf{S}$ to $\bot$. The only premise of the rule is the

$$(\textsc{crash})\ \frac{\mathbf{S}(i) \neq \bot}{\langle\, \mathbf{B}, \mathbf{Q}, \mathbf{S} \,\rangle \ \rightarrow \langle\, \mathbf{B}, \mathbf{Q}, \mathbf{S}\{\, i \mapsto \bot \,\} \,\rangle}$$

$$(\textsc{Qpp\_snd})\ \frac{\mathbf{S}(j) \neq \bot \qquad X \in \{\, \mathsf{P1}^r_{j\rightarrow k}, \mathsf{P2}^r_{j\rightarrow k}, \mathsf{P3}^r_{j\rightarrow k} \,\} \quad \mathbf{Q}.X = (w, \text{outgoing})}{\langle\, \mathbf{B}, \mathbf{Q}, \mathbf{S} \,\rangle \ \rightarrow \langle\, \mathbf{B}, \mathbf{Q}\{\, X \mapsto (w, \text{transit}) \,\}, \mathbf{S} \,\rangle}$$

$$(\textsc{Qpp\_deliver})\ \frac{\mathbf{S}(k) \neq \bot \qquad X \in \{\, \mathsf{P1}^r_{j\rightarrow k}, \mathsf{P2}^r_{j\rightarrow k}, \mathsf{P3}^r_{j\rightarrow k} \,\} \quad \mathbf{Q}.X = (w, \text{transit})}{\langle\, \mathbf{B}, \mathbf{Q}, \mathbf{S} \,\rangle \ \rightarrow \langle\, \mathbf{B}, \mathbf{Q}\{\, X \mapsto (w, \text{received}) \,\}, \mathbf{S} \,\rangle}$$

Table 6.1: QPP-Message Transmission and Process Crash

state of the process being defined, regardless of the precise value. This allows us to model the fact that any non-crashed process can crash at any time. For simplicity reasons, each execution of (CRASH) affects only one process. Thus, if we want to model the contemporary crash of a whole set of processes, we write a sequence of (CRASH) actions involving one by one all processes of the set. If no other action takes place in between, from the point of view of the system, the sequence of crashes perfectly models the contemporary crash.

6.1.2 QPP-messaging

As explained in §5.1, the component $\mathbf{Q}$ is a *global* data structure that contains QPP-messages and their transmission status. A particular $\mathbf{Q}$ is associated to a particular instant in time referring to a particular run. And more precisely, an instantaneous $\mathbf{Q}$ contains all QPP-messages that have been sent (and possibly received) by the n processes up to that instant.

Since the CT algorithm works with quasi-reliable point-to-point channels in crash failure model systems, we know from §5.1.1 that it is enough to use three transition states for modeling the lifetime of QPP-messages in our formalism. Such states are:

outgoing: messages that, by the execution of some QPP_SEND statement, are put into an unordered buffer for *outgoing* messages at the sender site; the intuition is that such messages have not yet left the sender site, so a sender crash might still make them unavailable forever.

$\mathrm{transit}$**:** messages that have left the sender site, but that have not yet arrived at their destination; these messages are kept in the unordered buffer of the transmission medium; the intuition is that such messages have already left the sender site and are available in the network, so they can be received even in case of a sender crash;

$\mathrm{received}$**:** messages that are stored—corresponding to the execution of some QPP_RECEIVE statement—within an unordered buffer for *received* messages at the intended receiver site.

In our data structure **Q**, we model these transmission states via the respective tags $\mathrm{outgoing}$, $\mathrm{transit}$, and $\mathrm{received}$. Thus, we record the information related to each message through tuples of the form

$$(r, P, j, k, \mathit{contents}, \tau)$$

where $\tau \in \{\, \mathrm{outgoing}, \mathrm{transit}, \mathrm{received} \,\}$. As we have already said in §5.1.1, component **Q** is not associated to a single location since it contains QPP-messages in all of its transmission states:

- by including message data of the form $(\cdot, \mathrm{transit})$ it captures the *non-local* state of the point-to-point medium that is seen as a buffer for the messages in transit from one site to another;
- by including message data of the form $(\cdot, \mathrm{outgoing})$ and $(\cdot, \mathrm{received})$, it represents the *local* unordered buffers of outgoing/received messages at the sender/receiver site.

According to the usage patterns of QPP-messages in the Consensus algorithm, we may observe that in any round r and phase P, there is at most one message ever sent from process j to process k (see Lemma 8.1.2.3 of message uniqueness). Thus, it suffices to model the occurring QPP-messages with ordinary sets instead of multisets. This observation corroborates the hypothesis we made in §5.1.1 that the round number r, the sender j and the receiver k are enough to uniquely identify each message. Thus, in order to conveniently manipulate the data structure **Q**, we use "structured" *selectors* (representing the message's control information) of the form

$$P^r_{j \to k}$$

for $P \in \{\, \mathsf{P1}, \mathsf{P2}, \mathsf{P3} \,\}$, $r \in \mathbb{N}$, and $j, k \in \mathbb{P}$. From the pseudo code, we see that processes never send QPP-messages in phase P4 (Table 3.1), so we do not need an entry for such phase. Using structured selectors, we extract data from entries in $\mathbf{Q}$ using the notation:

$$\mathbf{Q}.P^{r}_{j\to k} \overset{\text{def}}{=} \begin{cases} (\mathit{contents}, \tau) & \text{if } (r, P, j, k, \mathit{contents}, \tau) \in \mathbf{Q} \\ \bot & \text{otherwise} \end{cases}$$

We insert/update entries in $\mathbf{Q}$ by the notation $\mathbf{Q}\{\, X \mapsto V \,\}$ for

$$X \in \{\, \mathsf{P1}^{r}_{j\to k}, \mathsf{P2}^{r}_{j\to k}, \mathsf{P3}^{r}_{j\to k} \,\}$$

and

$$V \in (((\mathbb{V} \times \mathbb{N}) \cup \{\mathrm{ack}, \mathrm{nack}\}) \times \{\mathrm{outgoing}, \mathrm{transit}, \mathrm{received}\})$$

defined by:

$$\mathbf{Q}\{\, X \mapsto V \,\}.P^{r}_{j\to k} \overset{\text{def}}{=} \begin{cases} V & \text{if } X = P^{r}_{j\to k} \\ \mathbf{Q}.P^{r}_{j\to k} & \text{otherwise} \end{cases}$$

Using this notation, we adapt to our configurations the rules already presented in §5.1.1, that model the lifetime of messages across transmission states for quasi-reliable point-to-point channels in crash failure model systems. Thus, we obtain the rules (QPP_SND) and (QPP_DELIVER) in Table 6.1. Once again, note that these rules are independent of the current components $\mathbf{B}$ and $\mathbf{S}$ (in fact, $\mathbf{S}$ is only used to ensure that the process involved in the sending has not crashed). This reflects the fact that the underlying transmission medium ships messages from node to node, independently of any algorithm running at the nodes of the system. Since the pseudo-code of §3.1.1 ignores message transmission details, the reader will not find in it any correspondence with these rules. However, such model of the network is one of the strong points of our formalization because it allows us to have a clear view on the messages and their states at any time of the execution.

Further analyzing the usage patterns of QPP-messages within the Consensus algorithm, we observe that selectors $P^{r}_{j\to k}$ are always of the form $\mathsf{P1}^{r}_{i\to \mathrm{crd}(r)}$, $\mathsf{P2}^{r}_{\mathrm{crd}(r)\to i}$, or $\mathsf{P3}^{r}_{i\to \mathrm{crd}(r)}$. Note the different role played by i in the various cases (sender in $\mathsf{P1}$ and $\mathsf{P3}$, receiver in $\mathsf{P2}$). Note also that one of the abstract place-holders j, k always denotes the coordinator $\mathrm{crd}(r)$ of the round r in which the message is sent. Thus, one among the abstract place-holders j, k is actually redundant—which one depends on the phase of the message—but we stick to the verbose notation $P^{r}_{j\to k}$ for the sake of clarity.

For the data part, we mostly use the array formulation of *contents*.

- $\mathbf{Q}.\mathsf{P1}^{r}_{i\to\mathrm{crd}(r)}$ has the form $((v,s),\tau)$, where (v,s) is the *proposal* of process i at round r;
- $\mathbf{Q}.\mathsf{P2}^{r}_{\mathrm{crd}(r)\to i}$ has the form $((v,s),\tau)$, where (v,s) is the *round proposal* sent by coordinator $\mathrm{crd}(r)$ to all processes $i \in \mathbb{P}$;
- $\mathbf{Q}.\mathsf{P3}^{r}_{i\to\mathrm{crd}(r)}$ has the form (a,τ) where $a \in \{\,\mathrm{ack}, \mathrm{nack}\,\}$ is the positive/negative acknowledgment of process i at round r.

Initially, $\mathbf{Q}_0 = \emptyset$, because we assume that no message has been sent yet.

6.1.3 Reliable Broadcast

As we have seen in §5.2, Reliable Broadcast is not influenced by the properties of the system (reliable or unreliable communication, crash or crash-recovery failures). The broadcast and delivery rules are part of the algorithm: both the preconditions for the broadcast and the effects of the delivering are expressed in the pseudo-code of §3.1.1. A broadcast can happen in phase P4, if the process is coordinator and if the number of acknowledging messages among all the received P3 messages is greater than the majority. The delivery of a broadcasted message forces the decision field of the delivering process to be initialized to the value specified in the message, while it has no other effect if such field is already different from null. For the Chandra-Toueg algorithm, the rules describing respectively the broadcast and the delivery of a message are the last two rules of Table 6.2: (PHS-4-SUCCESS) and (RBC-DELIVER).

The formalization of Reliable Broadcast for CT is the same one we have seen in §5.2. We model $\mathbf{B}$ as an unbounded array indexed by the round number at the moment of sending. So, at any time of a run and for any round number $r \in \mathbb{N}$, $\mathbf{B}(r)$ is either undefined ($\perp$) or it is a pair of the form (v, D) where $v \in \mathbb{V}$ is the broadcasted value, and $D \subseteq \mathbb{P}$ is the set of processes that have not yet delivered this message. Initially $\mathbf{B}_0 := \emptyset$, that is $\mathbf{B}_0(r) = \perp$ for all $r \in \mathbb{N}$. We insert/update entries in $\mathbf{B}$ by the notation $\mathbf{B}\{\, r \mapsto V \,\}$ for $r \in \mathbb{N}$ and $V \in (\mathbb{V} \times 2^{\mathbb{P}})$ defined by:

$$\mathbf{B}\{\, r \mapsto V \,\}(\hat{r}) \overset{\text{def}}{=} \begin{cases} V & \text{if } \hat{r} = r \\ \mathbf{B}(\hat{r}) & \text{otherwise} \end{cases}$$

6.2 Auxiliary Notation

In order to put our interest only on important components and/or special properties of sets of messages, we introduce here some additional notation that will help us keeping the algorithm description comprehensible. Thus, in the following, we define precise projections from $\mathbf{Q}$ onto selected substructures. Let $Q \subseteq \mathbf{Q}$, then

$$Q.P^r \stackrel{\text{def}}{=} \{\, x \in Q \mid x = (r, P, \cdot, \cdot, \cdot, \cdot) \,\}$$

is used to project out of a given structure Q the slice that describes just those elements that were exchanged in phase P of round r between arbitrary processes. Apart from this abbreviation, which works solely on the selector part of $\mathbf{Q}$'s elements, we also use some specific projections with respect to the data part.

$$\begin{aligned} \text{received}(Q) &\stackrel{\text{def}}{=} \{\, x \in Q \mid x = (\cdot, \cdot, \cdot, \cdot, \cdot, \text{received}) \,\} \\ \text{ack}(Q) &\stackrel{\text{def}}{=} \{\, x \in Q \mid x = (\cdot, \cdot, \cdot, \cdot, \text{ack}, \cdot) \,\} \end{aligned}$$

Note that all of these resulting slices are just subsets of Q, so it makes sense to permit the selector notation $Q.P^r_{j \to k}$ also for such slices of Q. These abbreviations will be used later on in the formalization of the Consensus algorithm to check their relation to the majority of processes.

On a lower level, we abbreviate the extraction of information out of 1st- and 2nd-phase message contents as follows: we use $\text{prop}(((v,s),t)) := (v,s)$ to extract the proposal of a history element, while $\text{stamp}(((v,s),t)) := s$ denotes just the stamp associated to the proposal.

In phase P2, the coordinator of a round selects the best proposal among a majority of 1st-phase proposals for that round. The best proposal is the one with the highest stamp. If there exist more than one proposal with the same stamp, different strategies can apply. We choose here to select the proposal sent by the process with the smallest identifier. Other strategies can be reduced to this particular one by appropriately reordering processes.

Definition 6.2.0.1 *Let $\langle\, \mathbf{B}, \mathbf{Q}, \mathbf{S} \,\rangle$ be a configuration. Let* min *and* max *denote the standard operators to calculate the minimum and maximum, respectively, of a set of integers. Let*

$$\text{highest}^r(\mathbf{Q}) \stackrel{\text{def}}{=} \max\{\; s \mid (\cdot, \cdot, \cdot, \cdot, (\cdot, s), \cdot) \in \text{received}(\mathbf{Q}).\mathsf{P1}^r \;\}$$

denote the greatest (i.e., most recent) stamp occurring in a proposal that was received in round r. Let

$$\text{winner}^r(\mathbf{Q}) \stackrel{\text{def}}{=} \min\{\; j \mid (\cdot, \cdot, j, \cdot, (\cdot, \text{highest}^r(\mathbf{Q})), \cdot) \in \text{received}(\mathbf{Q}).\mathsf{P1}^r \;\}$$

denote the process that contributes the winning proposal for round r in $\mathbf{Q}.\mathsf{P1}$. *Then,*

$$\mathrm{best}^r(\mathbf{Q}) \overset{\mathrm{def}}{=} \mathrm{prop}(\mathbf{Q}.\mathsf{P1}^r_{\mathrm{winner}^r(\mathbf{Q})\to\mathrm{crd}(r)})$$

denotes the respective "best proposal" for round r, meaning the received proposal with the highest stamp sent by the process with the smallest identifier.

In phase $\mathsf{P2}$, the coordinator process $\mathrm{crd}(r)$ also sends $\mathrm{best}^r(\mathbf{Q})$ as the round proposal for round r to all processes. When this happens, we denote the value component of the round proposal as follows.

Definition 6.2.0.2

$$\mathrm{val}^r(\mathbf{Q}) \overset{\mathrm{def}}{=} \begin{cases} v & \textit{if, for all } i \in \mathbb{P},\ \mathbf{Q}.\mathsf{P2}^r_{\mathrm{crd}(r)\to i} = ((v,\cdot),\cdot) \\ \bot & \textit{otherwise} \end{cases}$$

Note that $\mathrm{val}^r(\mathbf{Q})$ is always uniquely defined, thus it can also be seen as a function.

6.3 The Algorithm in Transition-rule Format

$$(\textsc{while})\ \frac{\mathbf{S}(i) = ((r,\mathsf{W}),b,\bot)}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{Q}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{l}\mathbf{B}\\ \mathbf{Q}\\ \mathbf{S}\{\ \ i \ \mapsto\ ((r{+}1,\mathsf{P1}),b,\bot)\ \ \}\end{array}\right\rangle}$$

$$(\textsc{phs-1})\ \frac{\mathbf{S}(i) = ((r,\mathsf{P1}),b,d) \qquad P \overset{\mathrm{def}}{=} \text{if } i = \mathrm{crd}(r) \text{ then } \mathsf{P2} \text{ else } \mathsf{P3}}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{Q}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{lrcll}\mathbf{B}\\ \mathbf{Q}\{ & \mathsf{P1}^r_{i\to\mathrm{crd}(r)} & \mapsto & (b,\mathrm{outgoing}) & \}\\ \mathbf{S}\{ & i & \mapsto & ((r,P),b,d) & \}\end{array}\right\rangle}$$

$$(\textsc{phs-2})\ \frac{\mathbf{S}(i) = ((r,\mathsf{P2}),b,d) \qquad |\,\mathrm{received}(\mathbf{Q}).\mathsf{P1}^r| \geq \lceil (n{+}1)/2 \rceil}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{Q}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{lrcll}\mathbf{B}\\ \mathbf{Q}\{ & \mathsf{P2}^r_{\mathrm{crd}(r)\to j} & \mapsto & (\mathrm{best}^r(\mathbf{Q}),\mathrm{outgoing}) & \}_{\forall 1\leq j\leq n}\\ \mathbf{S}\{ & i & \mapsto & ((r,\mathsf{P3}),b,d) & \}\end{array}\right\rangle}$$

(the table continues on the next page)

$$(\textsc{phs-3-trust})\ \frac{\begin{array}{c}\mathbf{S}(i) = ((r, \mathsf{P3}), b, d) \qquad \mathbf{Q}.\mathsf{P2}^{r}_{\mathrm{crd}(r)\to i} = ((v,s), \mathrm{received}) \\ P \overset{\mathrm{def}}{=} \text{ if } i = \mathrm{crd}(r) \text{ then } \mathsf{P4} \text{ else } \mathsf{W}\end{array}}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{Q}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{llll}\mathbf{B} & & & \\ \mathbf{Q}\{ & \mathsf{P3}^{r}_{i\to\mathrm{crd}(r)} & \mapsto & (\mathrm{ack}, \mathrm{outgoing}) \ \} \\ \mathbf{S}\{ & i & \mapsto & ((r,P),(v,r),d) \ \}\end{array}\right\rangle}$$

$$(\textsc{phs-3-suspect})\ \frac{\begin{array}{c}\mathbf{S}(i) = ((r, \mathsf{P3}), b, d) \\ i \neq \mathrm{crd}(r)\end{array} \qquad \mathbf{Q}.\mathsf{P2}^{r}_{\mathrm{crd}(r)\to i} \neq (\cdot, \mathrm{received})}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{Q}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{llll}\mathbf{B} & & & \\ \mathbf{Q}\{ & \mathsf{P3}^{r}_{i\to\mathrm{crd}(r)} & \mapsto & (\mathrm{nack}, \mathrm{outgoing}) \ \} \\ \mathbf{S}\{ & i & \mapsto & ((r,\mathsf{W}),b,d) \ \}\end{array}\right\rangle}$$

$$(\textsc{phs-4-fail})\ \frac{\mathbf{S}(i) = ((r, \mathsf{P4}), b, d) \qquad \begin{array}{rcl}|\,\mathrm{received}(\mathbf{Q}).\mathsf{P3}^{r}| & \geq & \lceil (n{+}1)/2\rceil \\ |\,\mathrm{ack}(\mathrm{received}(\mathbf{Q}).\mathsf{P3}^{r})| & < & \lceil (n{+}1)/2\rceil\end{array}}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{Q}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{llll}\mathbf{B} & & & \\ \mathbf{Q} & & & \\ \mathbf{S}\{ & i & \mapsto & ((r,\mathsf{W}),b,d) \ \}\end{array}\right\rangle}$$

$$(\textsc{phs-4-success})\ \frac{\mathbf{S}(i) = ((r, \mathsf{P4}), (v,s), d) \qquad |\,\mathrm{ack}(\mathrm{received}(\mathbf{Q}).\mathsf{P3}^{r})| \geq \lceil (n{+}1)/2\rceil}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{Q}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{llll}\mathbf{B}\{ & r & \mapsto & (v, \mathbb{P}) \ \} \\ \mathbf{Q} & & & \\ \mathbf{S}\{ & i & \mapsto & ((r,\mathsf{W}),(v,s),d) \ \}\end{array}\right\rangle}$$

$$(\textsc{Rbc-deliver})\ \frac{\begin{array}{c}\mathbf{S}(i) = (c, b, d) \qquad \mathbf{B}(r) = (v, D) \qquad i \in D \\ d' \overset{\mathrm{def}}{=} \text{ if } d = \bot \text{ then } (v,r) \text{ else } d\end{array}}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{Q}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{llll}\mathbf{B}\{ & r & \mapsto & (v, D\backslash\{i\}) \ \} \\ \mathbf{Q} & & & \\ \mathbf{S}\{ & i & \mapsto & (c,b,d') \ \}\end{array}\right\rangle}$$

Table 6.2: The Chandra-Toueg Consensus algorithm

Now, everything is in place to reformulate the Consensus algorithm in terms of transition rules. Recall from § 6.1 that the global initial state of the formalized Consensus algorithm is:

$$\langle\, \mathbf{B}_0, \mathbf{Q}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \rangle \quad \stackrel{\text{def}}{=} \quad \langle\; \emptyset, \emptyset, \{\; i \mapsto ((0,\mathsf{W}), (v_i, 0), \bot) \;\}_{i\in\mathbb{P}} \;\rangle \;.$$

The rules are given in Table 6.2. Intuitively, rule (WHILE) models the `while` condition of the algorithm; the rules (PHS-*) describe how processes evolve, *sequentially*, from one phase to the next, and from one round to the next; the rule (RBC-DELIVER) models the possibility of RBC-delivering a message that has previously been RBC-broadcast by some coordinator. Our verbose explanations below are intended to show in detail how closely these rules correspond to the pseudo code of §3.1.1. Finally, note that the rules of Table 6.2 are to be used together with the algorithm-independent rules of Table 6.1 on point-to-point message transmission and process crashes.

Every rule typically picks one of the processes, say i, and, driven by i's current program counter, checks whether the respective "firing conditions" are satisfied. By definition, the rules can only be applied to non-crashed processes i, i.e., when $\mathbf{S}(i) \neq \bot$.

A process can execute rule (WHILE) only while being in phase W and having not yet decided. A process is in phase W when the algorithm is first started and also, as we will see by examining the other rules, when a round finishes and a new one is ready to begin. The execution of rule (WHILE) affects only the state of the process, in particular its program counter: the round number is incremented "by one" and the phase becomes P1. A new round starts. This is the only rule that can be executed by a process that begins the algorithm.

Rule (PHS-1) implements phase P1 of Table 3.1. The process can execute this rule only when being in phase P1. The execution has the effect of creating an outgoing 1st-phase message directed to the coordinator of the round, and of incrementing the state of the process to a new phase P. The value of P depends on whether the process is the coordinator of the round or not, and it is equal respectively to P2 or P3. To optimize the presentation of the semantics, rule (PHS-1) is not exactly the representation of P1 as it already performs the coordinator check that in the pseudo code is only made in P2.

Rule (PHS-2) represents phase P2 of the algorithm in Table 3.1. As said above, the rule does not check for coordinator identity because this test is already performed in rule (PHS-1). Consequently, only the coordinator has the possibility of being in phase P2 of a round. In order to execute this rule, the process must be in phase P2 and must have received at least a majority of the 1st-phase messages that have been sent to it for its current round. The execution increments the phase of the process state to

P3, and creates n outgoing messages (each of them addressed to a different process) carrying the best proposal, as selected by the coordinator.

Phase P3 of Table 3.1 presents an `await` with two alternatives. To render these conditions in our transition system we need to introduce two different rules. Rule (PHS-3-TRUST) is the representation of P3 when $\text{received\_Qpp}.\mathsf{P2}^r \neq \emptyset$. This rule can be executed only when the process is in phase P3 and the 2nd-phase message sent by the coordinator for that round has been received. The execution of (PHS-3-TRUST) has the effect of creating an outgoing positively acknowledging 3rd-phase message directed to the coordinator of the current round, and moving the process to a new phase P. Once again, the value of P depends on whether the process is the current coordinator of the round or not, and it is equal respectively to P4 or W. As explained before for (PHS-1), the rule (PHS-3-TRUST) is not the exact representation of P3 for $\text{received\_Qpp}.\mathsf{P2}^r \neq \emptyset$ because it performs the check on the coordinator identity that in the pseudo code is deferred to P4. Rule (PHS-3-SUSPECT) is the representation of P3 when the boolean condition $\text{suspected}(\text{crd}(r))$ is true. For convenience, a coordinator is not allowed to suspect itself, so (PHS-3-SUSPECT) can be executed by any process, except the coordinator, that is in phase P3 and has not yet received its 2nd-phase message for that round. Once again, this rule is not the exact representation of P3 for $\text{received\_Qpp}.\mathsf{P2}^r \neq \emptyset$ because it (implicitly) performs here the check on the coordinator identity that in the pseudo code is deferred to P4.

Phase P4 of Table 3.1 contains a conditional to determine whether the process is the coordinator of the current round; as we have seen this condition is already taken into account in rules (PHS-3-TRUST) and (PHS-3-SUSPECT). Both rule (PHS-4-FAIL) and rule (PHS-4-SUCCESS) can be executed only if the process has received at least a majority of 3rd-phase messages for its current round; this models the condition

$$\texttt{await}\ \ |\,\text{received\_Qpp}.\mathsf{P3}^r\,| \geq \lceil (n{+}1)/2 \rceil .$$

The rule (PHS-4-SUCCESS) implements the case when

$$|\,\text{ack}(\text{received\_Qpp}.\mathsf{P3}^r)\,| \geq \lceil (n{+}1)/2 \rceil$$

while rule (PHS-4-FAIL) is for the opposite case. Rule (PHS-4-FAIL) can be executed if the process is in phase P4, and if it has received at least a majority of 3rd-phase messages for the current round, but only if there is *not* a majority of positive acknowledgments. The execution simply increments the phase of the process state to W, enabling a new round to start (if the conditions of rule (WHILE) are matched). On the contrary, rule (PHS-4-SUCCESS) can be executed if the process is in phase P4, and if it has received at least a majority of positively acknowledging 3rd-phase messages for the current round (this implies that the number of 3rd-phase messages received by the process in the current round is at least equal to $\lceil (n+1)/2 \rceil$). The execution of rule (PHS-4-SUCCESS) increments the phase of the process state to W, making it possible for a new round to start. Moreover, it adds to $\mathbf{B}$, in correspondence to the current round, a new element containing the belief of the process. This is the representation of RBC_BROADCAST$(i,$ *state*.belief$)$ of Table 3.1. The identity of the process that has inserted the element in $\mathbf{B}$ can be retrieved later (if needed) from the round number to which it is associated. In fact, in each round, the only process that can be in phase P4 and can therefore execute rule (PHS-4-SUCCESS) is the coordinator of the round.

Rule (RBC-DELIVER) is the representation of the `when` block in Table 3.1. It can be executed at any time by any process provided that there is an element in $\mathbf{B}$ that contains the process identifier. The execution of (RBC-DELIVER) subtracts the process identifier from the list of processes that still have to deliver the message. Moreover, if the process has not yet decided (d is still undefined) the execution forces the process to decide for the value carried in the delivered message.

Notation. All derivations generated by the rules in Table 6.2 (since they model a deterministic algorithm) as well as by rule (CRASH) of Table 6.1 are fully determined by (1) the source configuration $\langle\, \mathbf{B}, \mathbf{Q}, \mathbf{S} \,\rangle$, (2) the identifier i of the process that executes the action —note that each rule "touches" at most one state component— and (3) the name (RULE) of the rule that is applied. The only exceptions are the QPP-rules, where the choice of the processed message is non-deterministic. Accordingly, in statements about runs, we will occasionally denote computation steps by writing

$$\langle\, \mathbf{B}, \mathbf{Q}, \mathbf{S} \,\rangle \rightarrow_{i:(\text{RULE})} \langle\, \mathbf{B}', \mathbf{Q}', \mathbf{S}' \,\rangle\ .$$

6.4 Formalization of Properties

Now that we have formally expressed CT in our model, we need to specify the component properties that will allow us to prove the correctness of the algorithm.

6.4.1 State Properties

The only property that needs to be defined for processes is correctness. According to [12], a process is called *correct* in a given run, if it does not crash in that run. To formalize this notion in our model, we just need to check the definedness of the relevant state entry in all configurations of the given run.

Definition 6.4.1.1 (Correct Processes) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$. *We define*

$$\mathsf{Correct}(R) := \{\, i \in \mathbb{P} \mid \forall t \in T : \mathbf{S}_t(i) \neq \bot \,\} \,.$$

A process $i \in \mathsf{Correct}(R)$ *is called* correct in R.

6.4.2 Network Properties

As we have already explained in §5.1, to formally state the quasi-reliability property of the messaging service based on our model, we need to specify when precisely the message of interest appears in a run for the first time, i.e., when the intended primitive QPP_SEND is executed. There are several rules in our semantics that start the sending of a message. In contrast, there is only a single rule that corresponds to QPP_RECEIVE, namely rule (QPP-DELIVER).

Definition 6.4.2.1 (Quasi-Reliable Point-To-Point) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *denote a run.* R *satisfies* QPP-Reliability *if, for all processes* $j, k \in \mathsf{Correct}(R)$ *and time* $t > 0$, *whenever*

$$\langle\, \cdot, \mathbf{Q}_{t-1}, \cdot \,\rangle \;\rightarrow_{j:(\cdot)}\; \langle\, \cdot, \mathbf{Q}_t, \cdot \,\rangle$$

with $\mathbf{Q}_{t-1}.P^r_{j\to k} = \bot$ *and* $\mathbf{Q}_t.P^r_{j\to k} = (\cdot, \mathrm{outgoing})$, *for* $r > 0$ *and* $P \in \{\, \mathsf{P1}, \mathsf{P2}, \mathsf{P3} \,\}$, *then there is* $u > t$ *such that*

$$\langle\, \cdot, \mathbf{Q}_{u-1}, \cdot \,\rangle \;\rightarrow_{k:(\text{QPP-DELIVER})}\; \langle\, \cdot, \mathbf{Q}_u, \cdot \,\rangle$$

with $\mathbf{Q}_u.P^r_{j\to k} = (\cdot, \mathrm{received})$.

It turns out that, by construction, and because we do not have message loss, finite complete Consensus runs are always QPP-reliable in the sense that all QPP-messages exchanged between correct processes must have been received when the final state is reached.

6.4.3 Reliable Broadcast Properties

In §5.2.1 we have shown general rules for expressing the Reliable Broadcast properties. Here we need to adapt them to the formalization of the Chandra-Toueg algorithm. As we can see from §6.1.3 and from Table 6.2, the rule that triggers the broadcast is (PHS-4-SUCCESS), while the rule that delivers a message is (RBC-DELIVER). Thus, the definition of the Reliable Broadcast properties becomes as follows.

Definition 6.4.3.1 (Reliable Broadcast) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *denote a run. Then we define some of its properties:*

1. RBC-Validity: *for every* $i \in \mathsf{Correct}(R)$*, if there is* $t > 0$ *such that*
$$\langle\, \mathbf{B}_{t-1}, \cdot, \cdot \,\rangle \;\rightarrow_{i:(\text{PHS-4-SUCCESS})}\; \langle\, \mathbf{B}_t, \cdot, \cdot \,\rangle$$
with $\mathbf{B}_{t-1}(r) = \bot$ *and* $\mathbf{B}_t(r) = (v, \mathbb{P})$ *for some* $r > 0$*, then there is* $u > t$ *such that*
$$\langle\, \mathbf{B}_{u-1}, \cdot, \cdot \,\rangle \;\rightarrow_{i:(\text{RBC-DELIVER})}\; \langle\, \mathbf{B}_u, \cdot, \cdot \,\rangle$$
with $\mathbf{B}_{u-1}(r) = (v, D_{u-1})$, $\mathbf{B}_u(r) = (v, D_u)$ *and* $D_{u-1} = D_u \uplus \{i\}$.
2. RBC-Agreement: *for every* $i \in \mathsf{Correct}(R)$*, if there is* $t > 0$ *such that*
$$\langle\, \mathbf{B}_{t-1}, \cdot, \cdot \,\rangle \;\rightarrow_{i:(\text{RBC-DELIVER})}\; \langle\, \mathbf{B}_t, \cdot, \cdot \,\rangle$$
with $\mathbf{B}_{t-1}(r) = (v, D_{t-1})$, $\mathbf{B}_t(r) = (v, D_t)$ *and* $D_{t-1} = D_t \uplus \{i\}$ *for some* $r > 0$,
then, for every $j \in \mathsf{Correct}(R)$*, there is* $u > 0$ *such that*
$$\langle\, \mathbf{B}_{u-1}, \cdot, \cdot \,\rangle \;\rightarrow_{j:(\text{RBC-DELIVER})}\; \langle\, \mathbf{B}_u, \cdot, \cdot \,\rangle$$
with $\mathbf{B}_{u-1}(r) = (v, D_{u-1})$, $\mathbf{B}_u(r) = (v, D_u)$ *and* $D_{u-1} = D_u \uplus \{j\}$.
3. (Uniform) RBC-Integrity: *for every* $i \in \mathsf{Correct}(R)$*, if there are* $t_1, t_2 > 0$ *such that*
$$\langle\, \mathbf{B}_{t_1-1}, \cdot, \cdot \,\rangle \;\rightarrow_{i:(\text{RBC-DELIVER})}\; \langle\, \mathbf{B}_{t_1}, \cdot, \cdot \,\rangle$$
$$\langle\, \mathbf{B}_{t_2-1}, \cdot, \cdot \,\rangle \;\rightarrow_{i:(\text{RBC-DELIVER})}\; \langle\, \mathbf{B}_{t_2}, \cdot, \cdot \,\rangle$$
and, for some $r > 0$*, both*
   (a) $\mathbf{B}_{t_1-1}(r) = (v, D_{t_1-1})$, $\mathbf{B}_{t_1}(r) = (v, D_{t_1})$*, and* $D_{t_1-1} = D_{t_1} \uplus \{i\}$.
   (b) $\mathbf{B}_{t_2-1}(r) = (v, D_{t_2-1})$, $\mathbf{B}_{t_2}(r) = (v, D_{t_2})$*, and* $D_{t_2-1} = D_{t_2} \uplus \{i\}$.

   then $t_1 = t_2$ *and there is* $u < t_1$ *such that*
$$\langle\, \mathbf{B}_{u-1}, \cdot, \cdot \,\rangle \;\rightarrow_{i:(\text{PHS-4-SUCCESS})}\; \langle\, \mathbf{B}_u, \cdot, \cdot \,\rangle$$
with $\mathbf{B}_{u-1}(r) = \bot$ *and* $\mathbf{B}_u(r) = (v, \mathbb{P})$.

Note that, in order to specify that some RBC-delivery actually corresponds to a particular RBC-broadcast, we explicitly refer to the round r of the respective (preceding) execution of broadcast.

6.4.4 Failure Detectors Properties

As we have seen in §2.2.1, FD Ω is defined to satisfy the following property: there is a time $\hat{t}$ after which all processes always trust (i.e., do not suspect) the same correct process k. Now, starting from what we presented in §5.4, we formalize this property for the Chandra-Toueg algorithm by expressing that, whenever after time $\hat{t}$ some process i in round r suspects the coordinator of r, then this suspected process is different from the trusted process k.

Definition 6.4.4.1 (Failure Detection) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run. R satisfies the property Ω if there is $k \in \mathsf{Correct}(R)$ and there is $\hat{t} > 0$ such that for all $t > \hat{t}$, whenever*

$$\langle\, \cdot, \cdot, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{i:(\text{PHS-3-SUSPECT})}\; \langle\, \cdot, \cdot, \mathbf{S}_t \,\rangle$$

and $\mathbf{S}_t(i) = ((r, P), \cdot, \cdot)$, *then* $k \neq \mathrm{crd}(r)$.

6.4.5 Admissible Runs

Now that we have formally expressed all the properties of the underlying modules and given a formal meaning to the term ”correct process”, we put all the properties together to select those runs of Consensus where only a minority of processes may crash, and that are based on the proper functioning of both the communication and the failure detection services.

Definition 6.4.5.1 (Admissible runs) *A Consensus run* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *is called* admissible *if it satisfies* QPP*-Reliability, the three* RBC*-properties, the Ω-property and, in addition, exhibits a majority of correct processes, i.e.,* $\mathsf{Correct}(R) \geq \lceil (n+1)/2 \rceil$.

In Chapter 8, we will prove that all admissible runs of Consensus satisfy the properties of Validity, Agreement and Termination.

Chapter 7

Formalization of the Part Time Parliament Algorithm

In the previous chapter we have expressed the Chandra-Toueg consensus algorithm using our formalization that models algorithms and underlying abstractions through the use of transition rules. Up to this point, we believe that the results are quite satisfying. So now, to check whether the proposed formalization can describe also other Consensus algorithms, we apply our method to the Part Time Parliament algorithm, also known as Paxos.

We remind here that Paxos is designed to work in presence of lossy and replicating channels, so, for the sake of clarity, when speaking about the Paxos algorithm we will use the term message *reproduction* to indicate the fact that a process sends the same message more than once (even though, as we will see later, this never happens in the Paxos algorithm) and the term message *duplication* or *replication* to indicate the fact that the network spontaneously produces copies of a message that is passing through it.

As for CT, in order to provide a complete account of the behavior of the Paxos algorithm, we need to explicitly represent the communication network as well as all interactions with it, in particular the buffering of CH- and RBC-messages. Thus, once again, we provide a mathematical structure that describes the global idealized run-time system comprising all processes and the network. The structure also plays the role of a system history that keeps track of relevant information during computation.

We define the Paxos algorithm in terms of inference rules, that specify computation steps as transitions between configurations of the form:

$$\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S} \,\rangle \;\rightarrow\; \langle\, \mathbf{B}', \mathbf{C}', \mathbf{S}' \,\rangle \quad \text{or, in vertical notation:} \quad \left\langle \begin{array}{c} \mathbf{B} \\ \mathbf{C} \\ \mathbf{S} \end{array} \right\rangle \rightarrow \left\langle \begin{array}{c} \mathbf{B}' \\ \mathbf{C}' \\ \mathbf{S}' \end{array} \right\rangle$$

where $\mathbf{B}$ and $\mathbf{C}$ contain the (histories of) messages sent during the execution of the algorithm through the Reliable *B*roadcast service and through the Point-to-Point service, respectively, while $\mathbf{S}$ models the *state* of the n processes.

As we did previously, the symbol $\rightarrow^*$ denotes the reflexive-transitive closure of $\rightarrow$ (formally defined in Tables 7.1 and 7.2). A *run* of the Paxos algorithm is then represented by sequences

$$\langle\, \mathbf{B}_0, \mathbf{C}_0, \mathbf{S}_0^{(v_1 \ldots, v_n)} \,\rangle \;\rightarrow^*\; \langle\, \mathbf{B}, \mathbf{C}, \mathbf{S} \,\rangle \;\rightarrow^{\imath} \qquad\qquad \text{where } \imath \in \{\, *, \infty \,\}$$

where $\langle\, \mathbf{B}_0, \mathbf{C}_0, \mathbf{S}_0^{(v_1, \ldots, v_n)} \,\rangle$ denotes the initial configuration based on the values $v_1, \ldots, v_n$ initially proposed by the n processes in the system.

7.1 Configurations

Next, we explain in detail the three components of a *configuration* $\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S} \,\rangle$.

7.1.1 Process State

As we remarked already for CT, when representing the algorithm by a set of step-oriented transition rules, we inevitably lose the intuitive sequencing of statements in the algorithm description. Following Lamport's idea in the Appendix of [35], we correct this loss of information by introducing subsequent phases where the algorithm needs to jump after the application of any transition rule. In the case of Paxos, such phases are idle, trying, $\mathrm{waiting}$, $\mathrm{polling}$, done.

idle means that the process is doing nothing special and that it is waiting for a message to come or for a good time to initiate a new round.

trying means that the process is leader and has just started a new round.

$\mathrm{waiting}$ means that the process is leader, that it has sent *prepare request* messages to all the other processes and it is waiting for answers.

$\mathrm{polling}$ means that the process is leader, that it has sent *accept request* messages to all the other processes and it is waiting for answers.

done is a phase that is not present in Lamport's description. It means that the process has done its job and has broadcasted a decision value. We have introduced it to emphasize the fact that a process has succeeded.

A possible alternative to the use of phase done was to change the state of the process directly from $\mathrm{polling}$ to idle, thus indicating that the process had gone back to a "quiet" state. This would have been similar to what we did for CT: a process that succeeds goes to phase W, thus indicating that it is ready to start a new round. However, while in CT phase W is the last phase in which a process can be before passing to a new round, in Paxos idle is generally used to indicate that a process is "unemployed". Thus, if we do not use done, it becomes impossible to tell whether a process has or has not gone through all the other phases of a round by simply looking at its state.

Component $\mathbf{S}$ is defined as an array of n elements, where $\mathbf{S}(i)$, for $1 \leq i \leq n$, represents the state of process i. The state of a process is a tuple of the form $\big(a,\ r,\ p,\ b,\ g,\ d\big)$, where

- $a = (l, Ph)$ consists of a variable $l \in \mathbb{L} = \{\mathrm{yes}, \mathrm{no}\}$ telling if the process is leader or not, and a phase label
  $$Ph \in \mathbb{H} = \{\mathrm{idle}, \mathrm{trying}, \mathrm{waiting}, \mathrm{polling}, \mathrm{done}\}$$
  note that the only possibility for a process that is not leader is to be idle;
- r is the *request number*;
- p is the *prepare request number*;
- $b = (v, s)$ is the *belief* (always defined), consisting of a value estimate $v \in \mathbb{V}$ and a stamp $s \in \mathbb{N}$, representing the request number in which the belief in the value was adopted;
- $g = (u, \iota)$ consists of a variable $u \in \{\bot, \top\}$, indicating if the process is down or up, and an *incarnation number* $\iota \in \mathbb{N}$ that keeps track of how many times the process has crashed and successively recovered (note that only the current incarnation can perform actions);
- $d = (v, s)$ is the *decision* (may be undefined, i.e., $d = \bot$), consisting of a value $v \in \mathbb{V}$ and a stamp $s \in \mathbb{N}$, representing the request number for which value v has been broadcast by $\mathrm{lead}(s)$.

The request numbers of different processes belong to mutually disjoint sets. We indicate with $\mathbb{R}(i) \cup \{0\}$ the set containing all the request numbers that process i can use during the execution of the algorithm. So, we have that for all $i, j \in \mathbb{P}$, $\mathbb{R}(i) \cap \mathbb{R}(j) = \emptyset$, and $\bigcup_{i \in \mathbb{P}} \mathbb{R}(i) = \mathbb{N} \setminus \{0\}$.

We call $\mathrm{lead}(r) \in (\mathbb{N}, \mathbb{P})$ the function that, given a request number r, returns the process that "owns" r.

$$\mathrm{lead}(r) = i \ \ with \ \ r \in \mathbb{R}(i)$$

Every process i starts being up, with incarnation number equals 1, not being leader (and therefore in phase idle), with request number and prepare request number equal to 0, believing in its own value v_i tagged with stamp 0; the decision field is left undefined. Thus, for every process i, the initial state is:

$$\mathbf{S}_0^{(v_1,\ldots,v_n)}(i) = \big((\mathrm{no}, \mathrm{idle}), 0, 0, (v_i, 0), (\top, 1), \bot\big)$$

We write $\mathbf{S}\{\, i \mapsto V \,\}$ for $i \in \mathbb{P}$ and
$V \in ((\mathbb{L} \times \mathbb{H}) \times \mathbb{N} \times \mathbb{N} \times (\mathbb{V} \times \mathbb{N}) \times (\{\bot, \top\} \times \mathbb{N}) \times ((\mathbb{V} \times \mathbb{N}) \cup \{\bot\}))$ to insert/update entries in $\mathbf{S}$ as follows:

$$\mathbf{S}\{\, i \mapsto V \,\}(k) \ \stackrel{\mathrm{def}}{=} \ \begin{cases} V & \text{if } k = i \\ \mathbf{S}(k) & \text{otherwise} \end{cases}$$

In Table 7.1, with rule (CRASH), we model the crash of process i by setting the u component of the entry $\mathbf{S}(i)$ to $\bot$. With rule (RECOVERY), we model the recovery of process i by setting the u component of the entry $\mathbf{S}(i)$ to $\top$ and incrementing of one the incarnation number (here and further, the function $\mathrm{succ}(x)$ classically retrieves the natural successor of value x). A process that recovers, comes back not being leader and being idle. Note that these rules are defined such that any non-crashed process can crash at any time and vice versa. The only premises of the rules are the u component of the process state being defined (respectively undefined), and the decision field being undefined, regardless of the precise values of the other fields.

Leader Election While in CT processes proceed in sequential rounds, thus alternatively becoming coordinators and having the right to take initiatives, in Paxos processes are "elected" leaders, and only the leaders are allowed to throw and master rounds. Therefore, we need to model the election of a leader in our formalism. The actions of the algorithm are then conditioned by the process being a leader or not. In Table 7.1, with rule (LEADER-YES) we model the passage of a process from being a simple participant to being a leader. With rule (LEADER-NO) we model the inverse step. Both rules can be executed only if the process has not yet decided. Note that if a process crashes while being leader and then recovers, it comes back not being leader and it has to execute rule (LEADER-YES) if it wants to become leader again.

$$(\text{CRASH})\ \frac{\mathbf{S}(i) = \big(a, r, p, b, (\top, \iota), \bot\big)}{\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle \ \rightarrow \langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\{\, i \mapsto \big(a, r, p, b, (\bot, \iota), \bot\big)\,\}\,\rangle}$$

$$(\text{RECOVERY})\ \frac{\mathbf{S}(i) = \big(a, r, p, b, (\bot, \iota), \bot\big)}{\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle \ \rightarrow \langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\{\, i \mapsto \big((\text{no}, \text{idle}), r, p, b, (\top, \text{succ}(\iota)), \bot\big)\,\}\,\rangle}$$

$$(\text{CH\_SND})\ \frac{X \in \{\, \mathsf{P0}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3}\,\} \qquad \mathbf{C}.X^r_{j\to k} = (w, (\iota, 0, 0, 0)) \qquad \mathbf{S}(j) = (\cdot, \cdot, \cdot, \cdot, (\top, \iota), \cdot)}{\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle \ \rightarrow \langle\, \mathbf{B}, \mathbf{C}\{\, X^r_{j\to k} \mapsto (w, (\iota, 1, 0, 0))\,\}, \mathbf{S}\,\rangle}$$

$$(\text{CH\_DELIVER})\ \frac{X \in \{\, \mathsf{P0}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3}\,\} \qquad \mathbf{C}.X^r_{j\to k} = (w, (\alpha, \text{succ}(\beta), \gamma, \delta)) \qquad \mathbf{S}(k) = (\cdot, \cdot, \cdot, \cdot, (\top, \iota), \cdot) \qquad \delta < \iota}{\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle \ \rightarrow \langle\, \mathbf{B}, \mathbf{C}\{\, X^r_{j\to k} \mapsto (w, (\alpha, \beta, \gamma, \iota))\,\}, \mathbf{S}\,\rangle}$$

$$(\text{CH\_DUPL})\ \frac{X \in \{\, \mathsf{P0}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3}\,\} \qquad \mathbf{C}.X^r_{j\to k} = (w.(\alpha, \beta, \gamma, \delta)) \qquad \beta > 0}{\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle \ \rightarrow \langle\, \mathbf{B}, \mathbf{C}\{\, X^r_{j\to k} \mapsto (w, (\alpha, \text{succ}(\beta), \gamma, \delta))\,\}, \mathbf{S}\,\rangle}$$

$$(\text{CH\_LOSS})\ \frac{X \in \{\, \mathsf{P0}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3}\,\} \qquad \mathbf{C}.X^r_{j\to k} = (w, (\alpha, \text{succ}(\beta), \gamma, \delta))}{\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle \ \rightarrow \langle\, \mathbf{B}, \mathbf{C}\{\, X^r_{j\to k} \mapsto (w, (\alpha, \beta, \text{succ}(\gamma), \delta))\,\}, \mathbf{S}\,\rangle}$$

$$(\text{LEADER-YES})\ \frac{\mathbf{S}(i) = \big((\text{no}, \text{idle}),\ r,\ p,\ b,\ (\top, \iota),\ \bot\big)}{\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle \ \rightarrow \langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\{\, i \mapsto \big((\text{yes}, \text{idle}),\ \text{next}_i(r),\ p,\ b,\ (\top, \iota),\ \bot\big)\,\}\,\rangle}$$

$$(\text{LEADER-NO})\ \frac{\mathbf{S}(i) = \big((\text{yes}, \cdot),\ r,\ p,\ b,\ (\top, \iota),\ \bot\big)}{\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle \ \rightarrow \langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\{\, i \mapsto \big((\text{no}, \text{idle}),\ r,\ p,\ b,\ (\top, \iota),\ \bot\big)\,\}\,\rangle}$$

Table 7.1: Process Crash/Recovery – CH-Message Transmission – Leader Election

7.1.2 CH-messaging

The component **C** is a *global* data structure that contains CH-messages and their transmission status. The idea is that a particular **C** is associated to a particular instant in time referring to a particular run. More precisely, an instantaneous **C** provides access to all the CH-messages that have been sent (and possibly received, duplicated or lost) by the n processes up to that instant.

If we look carefully at the CT pseudo-code and at the natural language description of Paxos, we notice that the messages exchanged between the leader and the other processes in Paxos strongly resemble the messages exchanged between the coordinator and the other processes in CT. In particular, we can draw the following correspondence:

the Paxos answer to a *prepare request* is the same as the CT message P1, since both of them are sent to the process mastering the round and contain the current belief of the sending process;

the Paxos *accept request* is the same as the CT message P2, since both of them are sent by the process mastering the round and contain the best among the beliefs received by the leader/coordinator;

the Paxos answer to an *accept request* is the same as an acknowledging CT message P3, since both of them are sent to the process mastering the round and contain a positive confirmation of message reception.

Therefore, for the sake of uniformity, we use in the formalization of Paxos the same message names we used for CT. The only message that appears in Paxos but does not exist in CT is the *prepare request*. Since this is the message that awakens all the sending of further messages in the round, we call it P0.

Since Paxos is designed to operate with point-to-point lossy and replicating channels in crash-recovery failure models, we have seen in §5.1.2 that messages pass through four different transmission states:

outgoing: messages that, by the execution of some SEND statement, are put into an unordered buffer for *outgoing* messages at the sender site; the intuition is that such messages have not yet left the sender site, so a sender crash might still make them unavailable forever;

transit: messages that have left the sender site, but that have not yet arrived at their destination; these messages are kept in the unordered buffer of the transmission medium and are not affected in case of a sender crash, but they can instead be replicated or lost;

lost**:** messages that are stored within an unordered buffer for *lost* messages in a sort of "black hole" in the transmission medium.

received**:** messages that are stored—corresponding to the execution of some RECEIVE statement—within an unordered buffer for *received* messages at the intended receiver site; a message can be received more than once if it has been replicated during its transit in the transmission medium;

In §5.1.2 we have also seen that we need a four components tuple in order to model the lifetime of CH-messages in our formalism. And that we can play with the components of this tuple to express the relationship between the incarnation number of a process and the messages sent and received by the process during its current incarnation.

We recap here the meaning and the possible values of each component. The first component, α, represents the incarnation number of the message sender at the time in which it inserts the message in its outgoing buffer. The second component, β, represents the number of copies of the message that are in transit in the network. The third component, γ, represents the number of copies of the message that have fallen into the "black hole" and are lost. The fourth component, δ, represents the incarnation number of the message receiver at the time in which it inserts the message in its received buffer. Since the sender of a message does not have any knowledge about the incarnation number of other processes, this component is initialized with the value 0 at the moment of creation of the message, and changes its value only upon reception. The same message can be received more than once, but only if the current incarnation number of the receiver is different from (greater than) the incarnation in which the message has been previously received. In this case, the current incarnation number overwrites the previous one. Doing this we loose information about how many times the message has been received up to that point in time. But since a process can access only the messages that are in its receiver buffer for the current incarnation, and since in our proofs we will take into account only the presence of messages without caring for the incarnation number of the receiver, we will actually experience no loss of meaningful information.

Thus, in our data structure **C**, we record the information related to each message through tuples of the form

$$(r, P, j, k, \mathit{contents}, \tau)$$

where $\tau = (\alpha, \beta, \gamma, \delta)$ and $\alpha, \beta, \gamma, \delta \in \mathbb{N}$. Note once again that component **C** is not associated to a single location in our Paxos system since it contains CH-messages in all transmission states.

According to the usage patterns of CH-messages in the algorithm, we may observe that in any request r and phase P, there is at most one message ever sent from process j to process k (see Lemma 9.1.2.3 of message uniqueness). Thus, it suffices to model the occurring CH-messages with ordinary sets instead of multisets. Once again, this observation corroborates the hypothesis we made in §5.1.1 that the round number r, the sender j and the receiver k are enough to uniquely identify each message. Thus, in order to conveniently manipulate the data structure $\mathbf{C}$, we use "structured" *selectors* (representing the message's control information) of the form

$$P^r_{j\to k}$$

for $P \in \{\,\mathsf{P0}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3}\,\}$, $r \in \mathbb{N}$, and $j, k \in \mathbb{P}$. Using structured selectors, we extract data from entries in $\mathbf{C}$ using the notation:

$$\mathbf{C}.P^r_{j\to k} \overset{\text{def}}{=} \begin{cases} (\mathit{contents}, \tau) & \text{if } (r, P, j, k, \mathit{contents}, \tau) \in \mathbf{C} \\ \bot & \text{otherwise} \end{cases}$$

We insert/update entries in $\mathbf{C}$ by the notation $\mathbf{C}\{\, X \mapsto V\,\}$ for $X \in \{\,\mathsf{P0}^r_{j\to k}, \mathsf{P1}^r_{j\to k}, \mathsf{P2}^r_{j\to k}, \mathsf{P3}^r_{j\to k}\,\}$ and
$V \in ((\mathbb{N} \cup (\mathbb{V} \times \mathbb{N}) \cup \{\mathbb{V}\}) \times (\mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}))$ defined by:

$$\mathbf{C}\{\, X \mapsto V\,\}.P^r_{j\to k} \overset{\text{def}}{=} \begin{cases} V & \text{if } X = P^r_{j\to k} \\ \mathbf{C}.P^r_{j\to k} & \text{otherwise} \end{cases}$$

Using this notation, we adapt to our configurations the rules already presented in §5.1, that model the network module in case of lossy and replicating channels and crash-recovery failure model. Thus we obtain the rules (CH_SND), (CH_DUPL), (CH_LOSS) and (CH_DELIVER) in Table 7.1 model the lifetime of messages across transmission states. As we said earlier, that these rules are independent of the current components $\mathbf{B}$ and $\mathbf{S}$ (in fact, $\mathbf{S}$ is only used to ensure that the process involved in the sending is not crashed and to register the incarnation number of the receiving process). This reflects the fact that the underlying transmission medium ships messages from node to node, duplicates and loses them independently of any algorithm running at the nodes of the system.

Further analyzing the usage patterns of CH-messages within the Paxos algorithm, we observe that selectors $P^r_{j\to k}$ are always of the form $\mathsf{P0}^r_{\text{lead}(r)\to i}$, $\mathsf{P1}^r_{i\to\text{lead}(r)}$, $\mathsf{P2}^r_{\text{lead}(r)\to i}$, or $\mathsf{P3}^r_{i\to\text{lead}(r)}$. Note the different role played by i in the various cases (receiver in P0 and P2, sender in P1 and P3). Note also that one of the abstract place-holders j, k always denotes the owner

$\mathrm{lead}(r)$ of the request r in which the message is sent. Thus, one of the abstract place-holders j, k is actually redundant—which one depends on the phase of the message—but we stick to the verbose notation $P^r_{j\to k}$ for the sake of clarity.

For the data part, we mostly use the array formulation of *contents*.

- $\mathbf{C}.\mathsf{P0}^r_{\mathrm{lead}(r)\to i}$ has the form (r, τ), where r is the next *request number* of process i;
- $\mathbf{C}.\mathsf{P1}^r_{i\to\mathrm{lead}(r)}$ has the form $((v, s), \tau)$, where (v, s) is the *proposal* of process i at request r;
- $\mathbf{C}.\mathsf{P2}^r_{\mathrm{lead}(r)\to i}$ has the form $((v, s), \tau)$, where (v, s) is the *request proposal* of the leader $\mathrm{lead}(r)$;
- $\mathbf{C}.\mathsf{P3}^r_{i\to\mathrm{lead}(r)}$ has the form (v, τ) where $v \in \mathbb{V}$ is the proposal for request r that the participant wants to acknowledge.
  Note that, since the leader does not store the request proposal value and it can or cannot acknowledge its own proposal, we need the processes to recall the leader the request proposal value. In this way, the leader can broadcast such value if it receives enough acknowledgments.

Initially, $\mathbf{C}_0 = \emptyset$, because we assume that no message has been sent yet.

7.1.3 Reliable Broadcast

As explained in §5.2, Reliable Broadcast does not depend on the properties of the system (reliable or unreliable communication, crash or crash-recovery failures). The broadcast conditions and the delivery effects of Paxos are specified inside the description of the algorithm in §3.2. A broadcast can happen when the process is leader, it is in phase $\mathrm{polling}$ and it has received a response to its last *accept request* from a majority of processes (step 5 of the algorithm description of §3.2.1). The delivery of a broadcasted message can happen at any time. It forces the decision field of the delivering process to be initialized to the value specified in the message (step 6 of the algorithm description of §3.2.1). Since it is not explicitly specified what happens in case of delivery of a message by a process whose decision field has already been initialized, we assume (like in CT) that such action has no effect in this case. As we will see with Lemma 9.2.2.3, our design choice is not really important since every broadcast message contains exactly the same value. For the Paxos algorithm, the rules describing respectively the broadcast and the delivery of a message are the last two rules of Table 7.2: (SUCCESS) and (RBC-DELIVER).

Note that the Paxos algorithm uses Reliable Broadcast exclusively for the dissemination of decisions: for any given request, only the leader that owns the corresponding request number can execute RBC_BROADCAST to trigger the sending of messages that mention it. Therefore, we use request numbers to properly distinguish all the broadcasts that may possibly occur in a single run. We model $\mathbf{B}$ as an unbounded array indexed by the request number $r \in \mathbb{N}$. At any time of a run and for any request number r, $\mathbf{B}(r)$ is either undefined ($\perp$) or a pair of the form (v, D) where $v \in \mathbb{V}$ is the decision value broadcasted by $\mathrm{lead}(r)$, and $D \subseteq \mathbb{P}$ is the set of processes that have not yet executed RBC_DELIVER for this message. Initially $\mathbf{B}_0 := \emptyset$, that is $\mathbf{B}_0(r) = \perp$ for all $r \in \mathbb{N}$. We insert/update entries in $\mathbf{B}$ by the notation $\mathbf{B}\{\, r \mapsto V \,\}$ for $r \in \mathbb{N}$ and $V \in (\mathbb{V} \times 2^{\mathbb{P}})$ defined by:

$$\mathbf{B}\{\, r \mapsto V \,\}(\hat{r}) \overset{\text{def}}{=} \begin{cases} V & \text{if } \hat{r} = r \\ \mathbf{B}(\hat{r}) & \text{otherwise} \end{cases}$$

7.2 Auxiliary Notation

As we have done for CT, we define here some auxiliary notation that will help us keeping more comprehensible both the description of the algorithm and the discussion on its properties.

Following the philosophy of indicating with a dot $\cdot$ any expression for which the actual value is not relevant in the context, when neither the process nor its incarnation are of any interest to us in the context, we indicate the pair $(\cdot, \cdot)$ with a simple $\cdot$.

In the following, we define precise projections from $\mathbf{C}$ onto selected sub-structures. Let $Q \subseteq \mathbf{C}$, then

$$Q.P^r \overset{\text{def}}{=} \{\, x \in Q \mid x = (r, P, \cdot, \cdot, \cdot, \cdot) \,\}$$

is used to project out of a given structure Q the slice that describes just those elements that are still available and that were exchanged in phase P of request r between arbitrary processes. Moreover,

$$Q.P^r_\iota \overset{\text{def}}{=} \{\, x \in Q.P^r \mid x = (r, P, \cdot, (\cdot, \iota), \cdot, \cdot) \,\}$$

is used to project out of $Q.P^r$ only the slice that contains those elements whose receiver has in incarnation number ι. Apart from this abbreviation, which works solely on the selector part of $\mathbf{C}$'s elements, we also use some specific projections with respect to the data part.

$$\begin{aligned}\mathrm{received}(Q) &\overset{\mathrm{def}}{=} \{\, x \in Q \mid x = (\cdot,\cdot,\cdot,\cdot,\cdot,(\cdot,\cdot,\cdot,\delta)) \wedge \delta \geq 1 \,\} \\ \mathrm{ack}_v(Q) &\overset{\mathrm{def}}{=} \{\, x \in Q \mid x = (\cdot,\mathsf{P3},\cdot,\cdot,v,\cdot) \,\}\end{aligned}$$

Note that all of these resulting slices are just subsets of $\mathbf{C}$, so it makes sense to permit the selector notation $\mathbf{C}.P^r_{j\to k}$ also for such slices of $\mathbf{C}$. These abbreviations will be used later on in the formalization of the Paxos algorithm to check their relation to the majority of processes.

On a lower level, we abbreviate the extraction of information out of 1st- and 2nd-phase message contents as follows: we use $\mathrm{prop}((v,s),\tau) := (v,s)$ to extract the proposal of a history element, while $\mathrm{stamp}((v,s),\tau) := s$ denotes just the stamp associated to the proposal.

In phase $\mathrm{waiting}$, the owner of a request selects the best proposal among a majority of answers to its *prepare request*. The best proposal is the one with the highest stamp. However, if there exist more than one proposal with the same stamp, different strategies can apply. As we did for CT, we choose here to select the proposal sent by the process with the smallest identifier. Other strategies can be reduced to this particular one by appropriately reordering processes.

Definition 7.2.0.1 *Let $\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S} \,\rangle$ be a configuration. Let* $\min$ *and* $\max$ *denote the standard operators to calculate the minimum and maximum, respectively, of a set of integers. Let*

$$\mathrm{highest}^r_\iota(\mathbf{C}) \overset{\mathrm{def}}{=} \max\{\ s \mid (\cdot,\cdot,\cdot,\cdot,(\cdot,s),\cdot) \in \mathrm{received}(\mathbf{C}).\mathsf{P1}^r_\iota\ \}$$

denote the greatest (i.e., most recent) stamp occurring in a proposal that was received for request r by a process having incarnation number ι. Let

$$\mathrm{winner}^r_\iota(\mathbf{C}) \overset{\mathrm{def}}{=} \min\{\ j \mid (\cdot,\cdot,(j,\cdot),\cdot,(\cdot,\mathrm{highest}^r_\iota(\mathbf{C})),\cdot) \in \mathrm{received}(\mathbf{C}).\mathsf{P1}^r_\iota\ \}$$

denote the process that contributes the winning proposal for request r in $\mathbf{C}.\mathsf{P1}$. Then,

$$\mathrm{best}^r_\iota(\mathbf{C}) \overset{\mathrm{def}}{=} \mathrm{prop}(\mathbf{C}.\mathsf{P1}^r_{\mathrm{winner}^r_\iota(\mathbf{C})\to\mathrm{lead}(r)})$$

denotes the respective "best proposal" for request r, meaning the received proposal with the highest stamp sent by the process with the smallest identifier to the leader with incarnation number ι.
Finally, let

$$\mathrm{next}_i(r) \overset{\mathrm{def}}{=} \min\{\ \hat{r} \mid \hat{r} \in \mathbb{R}(i) \wedge r < \hat{r}\ \}$$

be the next request number usable by process i.

In phase $\mathrm{waiting}$, the leader process $\mathrm{lead}(r)$ also sends $\mathrm{best}^r_\iota(\mathbf{C})$ as the request proposal for request r to all processes. When this happens, we denote the value component of the request proposal as follows.

Definition 7.2.0.2

$$\mathrm{val}^r(\mathbf{C}) \overset{\mathrm{def}}{=} \begin{cases} v & \textit{if, for all } i \in \mathbb{P}, \mathbf{C}.\mathsf{P2}^r_{\mathrm{lead}(r)\to i} = ((v,\cdot),\cdot,\cdot) \\ \bot & \textit{otherwise} \end{cases}$$

Note that $\mathrm{val}^r(\mathbf{C})$ is always uniquely defined, thus it can also be seen as a function.

7.3 The algorithm in Inference-rule Format

$$(\textsc{new})\ \frac{\mathbf{S}(i) = ((\mathrm{yes},\cdot), r, p, b, (\top,\iota), \bot)}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{C}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{llll}\mathbf{B} & & & \\ \mathbf{C} & & & \\ \mathbf{S}\{ & i & \mapsto & ((\mathrm{yes},\mathrm{trying}), \mathrm{next}_i(r), p, b, (\top,\iota), \bot) \ \}\end{array}\right\rangle}$$

$$(\textsc{snd-prep-req})\ \frac{\mathbf{S}(i) = ((\mathrm{yes},\mathrm{trying}), r, p, b, (\top,\iota), d)}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{C}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{llll}\mathbf{B} & & & \\ \mathbf{C}\{ & \mathsf{P0}^r_{i\to j} & \mapsto & (r, (\iota, 0, 0, 0)) \ \}_{\forall j \in \mathbb{P}} \\ \mathbf{S}\{ & i & \mapsto & ((\mathrm{yes},\mathrm{waiting}), r, p, b, (\top,\iota), d) \ \}\end{array}\right\rangle}$$

$$(\textsc{rcv-prep-req})\ \frac{\begin{array}{cc}\mathbf{S}(i) = (a, r_i, p, b, (\top,\iota), d) & \mathbf{C}.\mathsf{P0}^r_{j\to i} = (r, (\cdot,\cdot,\cdot,\delta)) \\ \delta \geq 1 & p < r\end{array}}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{C}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{llll}\mathbf{B} & & & \\ \mathbf{C}\{ & \mathsf{P1}^r_{i\to j} & \mapsto & (b, (\iota, 0, 0, 0)) \ \} \\ \mathbf{S}\{ & i & \mapsto & (a, r_i, r, b, (\top,\iota), d) \ \}\end{array}\right\rangle}$$

$$(\textsc{snd-acc-req})\ \frac{\begin{array}{c}\mathbf{S}(i) = ((\mathrm{yes},\mathrm{waiting}), r, p, b, (\top,\iota), d) \\ |\,\mathrm{received}(\mathbf{C}).\mathsf{P1}^r_\iota| \geq \lceil (n{+}1)/2 \rceil\end{array}}{\left\langle \begin{array}{c}\mathbf{B}\\ \mathbf{C}\\ \mathbf{S}\end{array}\right\rangle \to \left\langle \begin{array}{llll}\mathbf{B} & & & \\ \mathbf{C}\{ & \mathsf{P2}^r_{i\to j} & \mapsto & (\mathrm{best}^r_\iota(\mathbf{C}), (\iota, 0, 0, 0)) \ \}_{\forall j \in \mathbb{P}} \\ \mathbf{S}\{ & i & \mapsto & ((\mathrm{yes},\mathrm{polling}), r, p, b, (\top,\iota), d) \ \}\end{array}\right\rangle}$$

(the table continues on the next page)

$$(\textsc{rcv-acc-req})\ \frac{\mathbf{S}(i) = (a, r, p, (\cdot, s), (\top, \iota), d) \qquad \mathbf{C}.\mathsf{P2}^{p}_{j\to i} = ((v,\cdot),(\cdot,\cdot,\cdot,\delta)) \\ \delta \geq 1 \qquad s < p}{\left\langle \begin{array}{c} \mathbf{B} \\ \mathbf{C} \\ \mathbf{S} \end{array} \right\rangle \rightarrow \left\langle \begin{array}{llll} \mathbf{B} & & & \\ \mathbf{C}\{ & \mathsf{P3}^{p}_{i\to j} & \mapsto & (v, (\iota, 0, 0, 0)) \quad \} \\ \mathbf{S}\{ & i & \mapsto & (a, r, p, (v, p), (\top, \iota), d) \quad \} \end{array} \right\rangle}$$

$$(\textsc{success})\ \frac{\mathbf{S}(i) = ((\text{yes}, \text{polling}), r, p, b, (\top, \iota), d) \\ |\,\text{received}(\mathbf{C}).\mathsf{P3}^{r}_{\iota}| \geq \lceil (n{+}1)/2 \rceil \qquad \mathbf{C}.\mathsf{P3}^{r}_{j\to i} = (v, \cdot)}{\left\langle \begin{array}{c} \mathbf{B} \\ \mathbf{C} \\ \mathbf{S} \end{array} \right\rangle \rightarrow \left\langle \begin{array}{llll} \mathbf{B}\{ & r & \mapsto & (v, \mathbb{P}) \quad \} \\ \mathbf{C} & & & \\ \mathbf{S}\{ & i & \mapsto & ((\text{yes}, \text{done}), r, p, b, (\top, \iota), d) \quad \} \end{array} \right\rangle}$$

$$(\textsc{Rbc-deliver})\ \frac{\mathbf{S}(i) = (a, r, p, b, (\top, \iota), d) \qquad \mathbf{B}(r) = (v, D) \qquad i \in D \\ d' \stackrel{\text{def}}{=} \text{ if } d = \bot \text{ then } (v, r) \text{ else } d}{\left\langle \begin{array}{c} \mathbf{B} \\ \mathbf{C} \\ \mathbf{S} \end{array} \right\rangle \rightarrow \left\langle \begin{array}{llll} \mathbf{B}\{ & r & \mapsto & (v, D \backslash \{i\}) \quad \} \\ \mathbf{C} & & & \\ \mathbf{S}\{ & i & \mapsto & (a, r, p, b, (\top, \iota), d') \quad \} \end{array} \right\rangle}$$

Table 7.2: The Paxos algorithm

Everything is in place to reformulate the Paxos algorithm in terms of inference rules. Recall from § 7.1 that the global initial state of the formalized Paxos algorithm is:

$$\langle\, \mathbf{B}_0, \mathbf{C}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \rangle \ \stackrel{\text{def}}{=}\ \langle\ \emptyset, \emptyset, \{\ i \mapsto \big((\text{no}, \text{idle}), 0, 0, (v_i, 0), (\top, 1), \bot\big)\ \}_{i \in \mathbb{P}}\ \rangle\,.$$

The rules are given in Table 7.2. Intuitively, rule (NEW) models the fact that a leader can pick a new proposal number at any time; rules (SND-PREP-REQ), (RCV-PREP-REQ), (SND-ACC-REQ), (RCV-ACC-REQ), (SUCCESS) describe how processes evolve; rule (RBC-DELIVER) models the possibility of RBC-delivering a message that has previously been RBC-broadcast by some leader. Our verbose explanations below are intended to show in detail how closely these rules correspond to the description of the algorithm of §3.2.1. Finally, note that the rules of Table 7.2 are to be used together with the algorithm-independent rules of Table 7.1 on point-to-point message transmission, process crash/recovery and leader election.

Every rule typically picks one of the processes, say i, and checks whether the respective "firing conditions" are satisfied. By definition, the rules can only be applied to non-crashed processes i, i.e., when $\mathbf{S}(i) = (\cdot, \cdot, \cdot, \cdot, (\top, \cdot), \cdot)$. Also, process i with incarnation number ι can access only the messages that are marked with ι. All other messages are invisible to i.

A leader can execute rule (NEW) at any time, provided that it has not yet decided. Rule (NEW) sets the request number field of the process' state to the next number that the process is allowed to use to post its requests. The rule also sets the phase label of the process' state to trying, in order to make the process remember it has just started a new round.

When a leader is in phase trying, it can execute rule (SND-PREP-REQ). Such rule changes the phase of the process to waiting and creates n outgoing $\mathsf{P0}$ messages, one for each of the processes in $\mathbb{P}$. Together, rules (NEW) and (SND-PREP-REQ) represent the first step of the algorithm of §3.2.1 where *Proc* is taken to be the whole set of processes in the system ($\mathbb{P}$). We have chosen to split the first step of the algorithm into two separate rules in order to mark the difference between starting a new round and sending messages. This distinction also allows to send less messages in case a leader decides to start rounds without being willing to go through them.

Rule (RCV-PREP-REQ) implements step 2 of the algorithm. Upon the reception of a $\mathsf{P0}$ message from a leader, the process compares the *request number* present in the message with the value of the *prepare request* field in its state. If the former is greater than the latter, the process sends to the leader a message containing its current belief.

Rule (SND-ACC-REQ) implements step 3 of the algorithm. If a leader receives a number of $\mathsf{P1}$ messages, for its current *request number*, that is greater than the majority, it changes its phase to polling and sends out to all the processes in the system a message containing the best belief it has received. Note that we have decided to follow Lamport's idea of having a strict separation between the roles of the participants (proposers, acceptors, learners). The process executing this rule has the role of proposer and therefore does not change its estimate directly here, but it sends a message to all the acceptors (and therefore also to itself).

Rule (RCV-ACC-REQ) implements step 4 of the algorithm. When a process receives a $\mathsf{P2}$ message for the same *prepare request* it is waiting for, it changes its belief to the received one, using as stamp the current value of the *prepare request*. The process also sends back to the leader a sort of acknowledgment: a message containing once again the value that has been adopted. Sending the value, and not simply an empty message, is necessary because the leader, when playing the role of a proposer, does not have any memory of the values it has sent out. Note that, in order to avoid leaving the execution of this rule possible for more than once, we have added the constraint that the stamp field of the belief should be strictly smaller than the current *prepare request* number.

Rule (SUCCESS) can be executed by a leader in phase polling whenever it has received a majority of P3 messages for its current request number. The leader takes the value contained in one of the received messages and sends it out to all the other processes through a reliable broadcast. This rule implements step 5 of the algorithm, supposing that the *success* message is a special message that needs to be treated in a special way.

Rule (RBC-DELIVER) can be executed by a process whenever there is a broadcasted message that has not yet been delivered by the process itself. The execution forces the process to initialize its decision field (if it was not yet initialized). This rule implements step 6 of the algorithm, starting from the hypothesis that the success messages are sent via a special way.

Notation. Most derivations of computation steps $\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle \rightarrow \langle\, \mathbf{B}', \mathbf{C}', \mathbf{S}'\,\rangle$ are fully determined by (1) the source configuration $\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle$, (2) the identifier i of the process that executes the action —note that each rule "touches" at most one state component— and (3) the name (RULE) of the rule that is applied. This is true for all of the rules in Table 7.2 (since they model a deterministic algorithm) as well as for rules (CRASH), (RECOVERY), (LEADER-YES), (LEADER-NO) of Table 7.1. The only exceptions are the CH-rules, where the choice of the processed message is non-deterministic.

Accordingly, in statements about runs, we will occasionally denote computation steps by writing $\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle \rightarrow_{i:(\text{RULE})} \langle\, \mathbf{B}', \mathbf{C}', \mathbf{S}'\,\rangle$, or $\langle\, \mathbf{B}, \mathbf{C}, \mathbf{S}\,\rangle \rightarrow_{(\text{RULE})} \langle\, \mathbf{B}', \mathbf{C}', \mathbf{S}'\,\rangle$ if (RULE) is rule (CH_DUPL) or rule (CH_LOSS).

Chapter 8

Correctness Proofs for the Chandra-Toueg Algorithm

We present here the proofs of correctness for the Chandra and Toueg Consensus algorithm. We start by presenting a set of lemmas that allow us to factor out some properties of the algorithm to which we are be able to make reference later, when dealing with the more substantial proofs. We then approach the fundamental properties of Consensus, starting from the simple Validity, going through the more complex Agreement and finishing with the tricky Termination.

8.1 Basic Properties of the Algorithm

In this section we prove a number of basic properties of the Consensus algorithm defined in Section 6.3. Most of them are simple, but still very useful properties holding for (prefixes of) runs of the form:

$$\langle\, \mathbf{B}_0, \mathbf{Q}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \,\rangle \;\rightarrow^* \langle\, \mathbf{B}, \mathbf{Q}, \mathbf{S} \,\rangle\,.$$

When carrying out proofs more informally, on the basis of pseudo code, the properties of this section are those that would typically be introduced via the words ”by inspection of the code”. To express them formally, we first need to define some appropriate orderings on the components of configurations.

8.1.1 Orderings

Natural orders can be defined on process states as well as point-to-point and broadcast histories.

Phase identifiers are transitively ordered as $\mathsf{P1} < \mathsf{P2} < \mathsf{P3} < \mathsf{P4} < \mathsf{W}$; we write $\leq$ to denote the reflexive closure of $<$. From the order on phases it is easy to derive a partial order on program counters that takes into account whether the respective process, say i, is coordinator of a particular round r.

Definition 8.1.1.1 (Ordering Program Counters) *Let $r, r' \in \mathbb{N}$ and $P, P' \in \{\,\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{P4}, \mathsf{W}\,\}$.*

1. *We write $(r, P) \leq (r', P')$ if $r < r'$ or $(r = r' \wedge P \leq P')$.*
2. *The pair (r, P) is a* valid program counter *for some process i if:*
   (a) *$i \neq \mathrm{crd}(r)$ implies $P \in \{\,\mathsf{P1}, \mathsf{P3}, \mathsf{W}\,\}$, and*
   (b) *$r = 0$ implies $P = \mathsf{W}$.*
3. *We write $(r, P) \leq_i (r', P')$ if $(r, P) \leq (r', P')$ and both (r, P) and (r', P') are valid for process i.*

We write $(r, P) <_i (r', P')$ (respectively, $(r, P) < (r', P')$) if $(r, P) \leq_i (r', P')$ (respectively, $(r, P) \leq (r', P')$) and $(r, P) \neq (r', P')$.

Notice that in the definition of $\leq_i$ the validity conditions are necessary as only coordinators go through phases $\mathsf{P2}$ and $\mathsf{P4}$, and as round 0 starts in phase W. Due to the latter, there is no r and P such that $(r, P) <_i (0, \mathsf{W})$.

If we consider the partial order on program counters, together with the information on the processes' crash status (where defined is smaller than undefined) and the value of the decision field (where undefined is smaller than defined), then we can establish a partial order on both the global state array and the local states of the processes.

Definition 8.1.1.2 (Ordering States) *Let $\mathbf{S}$ and $\mathbf{S}'$ be process states and $i \in \mathbb{P}$. We write $\mathbf{S}(i) \leq \mathbf{S}'(i)$ if*

1. *either $\mathbf{S}'(i) = \perp$,*
2. *or $\mathbf{S}(i) = (c_i, \cdot, d_i)$ and $\mathbf{S}'(i) = (c'_i, \cdot, d'_i)$ where*
   - *either $c_i <_i c'_i$,*
   - *or $c_i = c'_i \;\wedge\; ((d_i = d'_i) \vee d_i = \perp)$.*

We write $\mathbf{S} \leq \mathbf{S}'$ if for all $i \in \mathbb{P}$ it holds that $\mathbf{S}(i) \leq \mathbf{S}'(i)$. We write $\mathbf{S}(i) < \mathbf{S}'(i)$ if $\mathbf{S}(i) \leq \mathbf{S}'(i)$ and $\mathbf{S}(i) \neq \mathbf{S}'(i)$. Similarly, we write $\mathbf{S} < \mathbf{S}'$ if $\mathbf{S} \leq \mathbf{S}'$ and $\mathbf{S} \neq \mathbf{S}'$.

Note that the global ordering on state arrays properly reflects the local character of the transition rules of §6.3. As we will show with Lemma 8.1.2.1 each computation step of the algorithm corresponds precisely to the state change of a single process.

Message histories are ordered as follows: an entry defined in the later history must either be undefined in the older history, or its tag in the older history is smaller according to the transitive order:

$$\text{outgoing} < \text{transit} < \text{received} \ .$$

Definition 8.1.1.3 (Ordering QPP-Message Histories) *Let* $\mathbf{Q}, \mathbf{Q}'$ *be QPP-message histories.*
Then $\mathbf{Q} \leq \mathbf{Q}'$ *iff for all* $X \in \{ \mathsf{P1}^r_{i\to\mathrm{crd}(r)}, \mathsf{P2}^r_{\mathrm{crd}(r)\to i}, \mathsf{P3}^r_{i\to\mathrm{crd}(r)} \}$*, with* $i \in \mathbb{P}$ *and* $r \in \mathbb{N}$ *:*

1. *either* $\mathbf{Q}.X = \perp$
2. *or* $\mathbf{Q}.X = (V, \tau) \ \wedge \ \mathbf{Q}'.X = (V, \tau') \ \wedge \ \tau \leq \tau'$.

As usual, we define the strict counterpart as $\mathbf{Q} < \mathbf{Q}'$ *iff* $\mathbf{Q} \leq \mathbf{Q}'$ *and* $\mathbf{Q} \neq \mathbf{Q}'$.

In fact, our transition rules (QPP_SND) and (QPP_DELIVER) overwrite transmission tags, but only in increasing order while leaving the other data unchanged (see Corollary 8.1.2.5). Note that the order of QPP-message histories is transitive.

Finally, broadcast histories are ordered according to the decreasing delivery set of their entries.

Definition 8.1.1.4 (Ordering RBC-Message Histories) *Let* $\mathbf{B}, \mathbf{B}'$ *be RBC-message histories and* $r \in \mathbb{N}$. *We write* $\mathbf{B}(r) \leq \mathbf{B}'(r)$ *iff*

1. *either* $\mathbf{B}(r) = \perp$
2. *or* $\mathbf{B}(r) = (v, D)$ *and* $\mathbf{B}'(r) = (v, D')$ *with* $D \supseteq D'$.

We write $\mathbf{B} \leq \mathbf{B}'$ *if for all* $r \in \mathbb{N}$ *it holds that* $\mathbf{B}(r) \leq \mathbf{B}'(r)$. *We write* $\mathbf{B}(r) < \mathbf{B}'(r)$ *if* $\mathbf{B}(r) \leq \mathbf{B}'(r)$ *and* $\mathbf{B}(r) \neq \mathbf{B}'(r)$. *Similarly, we write* $\mathbf{B} < \mathbf{B}'$ *if* $\mathbf{B} \leq \mathbf{B}'$ *and* $\mathbf{B} \neq \mathbf{B}'$.

As for QPP-message histories, overwriting of RBC-message histories may happen, but only decreasing the delivery set, while keeping the recorded broadcast value unchanged (see Lemma 8.1.2.4).

8.1.2 Monotonicity and Message Evolution

In every Consensus run, states and message histories are monotonically non-decreasing. We start from the monotonicity of the state component, which is derived from the respective program counters.

Lemma 8.1.2.1 (Monotonicity of States) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run. Let*

$$\langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle \;\rightarrow_{i:(\text{RULE})} \langle\, \mathbf{B}_u, \mathbf{Q}_u, \mathbf{S}_u \,\rangle$$

be an arbitrary computation step of R*, for some* $u \in T$*,* $i \in \mathbb{P}$*, with* $\mathbf{S}_{u-1}(i) = (c_{u-1}, \cdot, \cdot)$*.*

1. *If* (RULE) *is rule* (CRASH)*, then* $\mathbf{S}_u = \bot$*, and hence* $\mathbf{S}_{u-1}(i) < \mathbf{S}_u(i)$ *and* $\mathbf{S}_{u-1} < \mathbf{S}_u$*.*
2. *If* (RULE) *is any of the rules* (QPP_SEND) *or* (QPP_DELIVER)*, then* $\mathbf{S}_u(i) = (c_u, \cdot, \cdot)$ *with* $c_{u-1} = c_u$*, and hence* $\mathbf{S}_{u-1}(i) = \mathbf{S}_u(i)$*, and* $\mathbf{S}_{u-1} = \mathbf{S}_u$*.*
3. *If* (RULE) *is rule* (WHILE)*, then* $\mathbf{S}_u(i) = (c_u, \cdot, \cdot)$ *with* $c_{u-1} < c_u$*, and hence* $\mathbf{S}_{u-1}(i) < \mathbf{S}_u(i)$*, and* $\mathbf{S}_{u-1} < \mathbf{S}_u$*.*
4. *If* (RULE) *is any of the* (PHS-*)*-rules, then* $\mathbf{S}_u(i) = (c_u, \cdot, \cdot)$ *with* $c_{u-1} < c_u$*, and hence* $\mathbf{S}_{u-1}(i) < \mathbf{S}_u(i)$*, and* $\mathbf{S}_{u-1} < \mathbf{S}_u$*.*
5. *If* (RULE) *is rule* (RBC-DELIVER)*, then* $\mathbf{S}_u(i) = (c_u, \cdot, \cdot)$ *with* $c_{u-1} = c_u$*, and hence* $\mathbf{S}_{u-1}(i) \leq \mathbf{S}_u(i)$*, and* $\mathbf{S}_{u-1} \leq \mathbf{S}_u$*.*

Proof. Immediate from the local comparison of premises and conclusions of each individual rule (RULE), and Definition 8.1.1.2 of the state ordering derived from program counters. □

Proving the monotonicity of the QPP-message history $\mathbf{Q}$ and of the RBC-message history $\mathbf{B}$ takes instead a bit more effort (see Lemma 8.1.2.4). Analyzing the transition rules in Table 6.1 and 6.2, we see that there are precisely four rules that *add* messages to a component $\mathbf{Q}$, namely (PHS-1), (PHS-2), (PHS-3-TRUST), and (PHS-3-SUSPECT). Moreover, there are two rules that change the content of messages in a component $\mathbf{Q}$, namely (QPP_SND) and (QPP_DELIVER). All the other rules leave the component $\mathbf{Q}$ untouched. As for $\mathbf{B}$, there is only one rule that adds messages to it (PHS-4-SUCCESS). Moreover, the application of rule (RBC-DELIVER) changes the content of messages in a component $\mathbf{B}$ while all other rules leave $\mathbf{B}$ untouched.

The following lemma describes how 1st-, 2nd-, 3rd-phase and broadcast messages are created, essentially by reading the transition rules backwards and identifying appropriate conditions. More precisely, from the definedness of an entry in the QPP-message or in the RBC-message history of some reachable configuration, we can derive that there must have been along the way (starting from the initial configuration) a uniquely defined computation step, where this particular entry was first added. This addition must have been produced by one of the aforementioned five rules. In order to uniquely identify such steps, we decompose the run that led to the assumed reachable configuration into four parts, as follows. (We recall here that with $\mathbf{Q}_u.P^r_{i\to j} \neq \bot$ we mean that at time u the entry for message $P^r_{i\to j}$ is defined and has the form $(\mathit{contents}, \tau)$. This means that at time u, message $P^r_{i\to j}$ has been sent.)

Lemma 8.1.2.2 (Message Creation) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)^{(v_1,\dots,v_n)}_T$ *be a run. Then,*

1. *If* $\mathbf{Q}_u.\mathsf{P1}^r_{i\to\mathrm{crd}(r)} \neq \bot$, *for some* $u \in T$, $i \in \mathbb{P}$, *and* $r > 0$, *then there is* $t \in T$, $t \leq u$, *for which*

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \rightarrow_{i:(\text{PHS-1})} \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle$$

such that, for some $b \in \mathbb{V} \times \mathbb{N}$

$$\begin{array}{rclrcl} \mathbf{Q}_{t-1}.\mathsf{P1}^r_{i\to\mathrm{crd}(r)} & = & \bot & \mathbf{S}_{t-1}(i) & = & \big((r,\mathsf{P1}), b, \cdot\big) \\ \mathbf{Q}_{t}.\mathsf{P1}^r_{i\to\mathrm{crd}(r)} & = & (b, \text{outgoing}) & \mathbf{S}_{t}(i) & = & \big((r,P), b, \cdot\big) \end{array}$$

where, if $i = \mathrm{crd}(r)$ *then* $P = \mathsf{P2}$, *otherwise* $P = \mathsf{P3}$.

2. *If* $\mathbf{Q}_u.\mathsf{P2}^r_{\mathrm{crd}(r)\to k} \neq \bot$, *for some* $u \in T$, $k \in \mathbb{P}$, *and* $r > 0$, *then there is* $t \in T$, $t \leq u$, *for which*

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \rightarrow_{\mathrm{crd}(r):(\text{PHS-2})} \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle$$

such that, for some $b \in \mathbb{V} \times \mathbb{N}$ *and for all* $i \in \mathbb{P}$

$$\begin{array}{ll} \mathbf{Q}_{t-1}.\mathsf{P2}^r_{\mathrm{crd}(r)\to i} = \bot & \mathbf{S}_{t-1}(\mathrm{crd}(r)) = \big((r,\mathsf{P2}), b, \cdot\big) \\ \mathbf{Q}_{t}.\mathsf{P2}^r_{\mathrm{crd}(r)\to i} = (\mathrm{best}^r(\mathbf{Q}_{t-1}), \text{outgoing}) & \mathbf{S}_{t}(\mathrm{crd}(r)) = \big((r,\mathsf{P3}), b, \cdot\big) \end{array}$$

3. *If* $\mathbf{Q}_u.\mathsf{P3}^r_{i\to\mathrm{crd}(r)} \neq \bot$, *for some* $u \in T$, $i \in \mathbb{P}$, *and* $r > 0$, *then there is* $t \in T$, $t \leq u$, *for which*

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \rightarrow_{i:(\text{RULE})} \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle$$

such that either of the following is true, for some $b \in \mathbb{V} \times \mathbb{N}$:

   (a) Rule (RULE) *is* (PHS-3-SUSPECT), $i \neq \mathrm{crd}(r)$ *and*

$$\begin{array}{ll} \mathbf{Q}_{t-1}.\mathsf{P3}^r_{i\to\mathrm{crd}(r)} = \bot & \mathbf{S}_{t-1}(i) = \big((r,\mathsf{P3}), b, \cdot\big) \\ \mathbf{Q}_{t}.\mathsf{P3}^r_{i\to\mathrm{crd}(r)} = (\text{nack}, \text{outgoing}) & \mathbf{S}_{t}(i) = \big((r,\mathsf{W}), b, \cdot\big) \end{array}$$

(b) *Rule* (RULE) *is* (PHS-3-TRUST) *and*

$$\mathbf{Q}_{t-1}.\mathsf{P3}^{r}_{i\to\mathrm{crd}(r)} = \bot \qquad \mathbf{S}_{t-1}(i) = \big((r,\mathsf{P3}), b, \cdot\big)$$

$$\mathbf{Q}_{t}.\mathsf{P3}^{r}_{i\to\mathrm{crd}(r)} = (\mathrm{ack}, \mathrm{outgoing}) \quad \mathbf{S}_{t}(i) = \big((r,P), (\mathrm{val}^r(\mathbf{Q}_{t-1}), r), \cdot\big)$$

where $\mathrm{val}^r(\mathbf{Q}_{t-1}) \neq \bot$ *and if* $i = \mathrm{crd}(r)$ *then* $P = \mathsf{P4}$, *otherwise* $P = \mathsf{W}$.

4. *If* $\mathbf{B}_u(r) \neq \bot$, *for some* $u \in T$ *and for some* $r > 0$, *then there is* $t \in T$, $t \leq u$, *for which*

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{\mathrm{crd}(r):(\text{PHS-4-SUCCESS})}\; \langle\, \mathbf{B}_{t}, \mathbf{Q}_{t}, \mathbf{S}_{t} \,\rangle$$

such that, for some $v \in \mathbb{V}$

$$\begin{array}{rclrcl} \mathbf{B}_{t-1}(r) & = & \bot & \mathbf{S}_{t-1}(\mathrm{crd}(r)) & = & \big((r,\mathsf{P4}), (v,\cdot), \cdot\big) \\ \mathbf{B}_{t}(r) & = & (v, \mathbb{P}) & \mathbf{S}_{t}(\mathrm{crd}(r)) & = & \big((r,\mathsf{W}), (v,\cdot), \cdot\big) . \end{array}$$

An important property of the Consensus algorithm is that both QPP- and RBC-messages are unique. This is guaranteed by the standard technique of adding enough distinguishing information to messages. Intuitively, we formalize the absence of message duplication by stating that *every addition of a message is fresh*. More precisely, whenever in a run a computation step is derived using one of the five rules that *add* a message to some component $\mathbf{Q}$ or $\mathbf{B}$, then the respective entry *must* have been undefined up to this very moment. An alternative and slightly more technical interpretation of this result is: if we had included conditions to ensure the prior non-definedness of added messages as premises to the respective rules, then those conditions would have been redundant.

Lemma 8.1.2.3 (Absence of Message Duplication) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)^{(v_1,\ldots,v_n)}_{T}$ *be a run such that, for some* $u \in T$,

$$\langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle \;\rightarrow_{i:(\text{RULE})}\; \langle\, \mathbf{B}_{u}, \mathbf{Q}_{u}, \mathbf{S}_{u} \,\rangle$$

with $\mathbf{S}_{u-1}(i) = ((r,P),\cdot,\cdot)$, *for some* $r \in \mathbb{N}$, $r > 0$, *and* $P \in \{\mathsf{P1}, \mathsf{P2}, \mathsf{P3}, \mathsf{P4}\}$. *Then,*

1. *If* (RULE) *is* (PHS-1), *then* $\mathbf{Q}_{u-1}.\mathsf{P1}^{r}_{i\to\mathrm{crd}(r)} = \bot$.
2. *If* (RULE) *is* (PHS-2), *then* $\mathbf{Q}_{u-1}.\mathsf{P2}^{r}_{\mathrm{crd}(r)\to k} = \bot$.
3. *If* (RULE) *is* (PHS-3-TRUST) *or* (PHS-3-SUSPECT), *then* $\mathbf{Q}_{u-1}.\mathsf{P3}^{r}_{i\to\mathrm{crd}(r)} = \bot$.
4. *If* (RULE) *is* (PHS-4-SUCCESS), *then* $\mathbf{B}_{u-1}(r) = \bot$.

Putting it all together, we can now state the monotonicity results for the two kinds of messages.

Lemma 8.1.2.4 (Monotonicity of Message Histories) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run such that*

$$\langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle \;\rightarrow_{i:(\text{RULE})}\; \langle\, \mathbf{B}_u, \mathbf{Q}_u, \mathbf{S}_u \,\rangle$$

for some $u \in T$ *and* $i \in \mathbb{P}$.

1. *If* (RULE) *is any of the rules* (WHILE), (PHS-4-FAIL), (PHS-4-SUCCESS), (RBC-DELIVER), (CRASH), *then* $\mathbf{Q}_{u-1} = \mathbf{Q}_u$.
2. *If* (RULE) *is any of the rules* (PHS-1), (PHS-2), (PHS-3-TRUST), (PHS-3-SUSPECT), (QPP_SND), *or* (QPP_DELIVER), *then* $\mathbf{Q}_{u-1} < \mathbf{Q}_u$.
3. *If* (RULE) *is any of the rules* (PHS-4-SUCCESS) *or* (RBC-DELIVER), *then* $\mathbf{B}_{u-1} < \mathbf{B}_u$.
4. *If* (RULE) *is any of the rules* (WHILE), (PHS-1), (PHS-2), (PHS-3-TRUST), (PHS-3-SUSPECT), (PHS-4-FAIL), (QPP_SND), (QPP_DELIVER), *or* (CRASH), *then* $\mathbf{B}_{u-1} = \mathbf{B}_u$.

Proof. We prove the result by local comparison of premises and conclusions of each possible rule (RULE).
If (RULE) is any of the rules (WHILE), (PHS-4-FAIL), (PHS-4-SUCCESS), (RBC-DELIVER), (CRASH) the $\mathbf{Q}$ component is not touched, and $\mathbf{Q}_{u-1} = \mathbf{Q}_u$.
Similarly, if (RULE) is any of the rules (WHILE), (PHS-1), (PHS-2), (PHS-3-TRUST), (PHS-3-SUSPECT), (PHS-4-FAIL), (QPP_SND), (QPP_DELIVER), or (CRASH), the $\mathbf{B}$ component is not touched, and $\mathbf{B}_{u-1} = \mathbf{B}_u$.
If (RULE) is (PHS-1), (PHS-2), (PHS-3-TRUST), or (PHS-3-SUSPECT), then we know (Lemma 8.1.2.3) that $\mathbf{Q}_{u-1}.X = \bot$, while the application of (RULE) produces $\mathbf{Q}_u.X \neq \bot$ with $X \in \{\, \mathsf{P1}^r_{j\to k}, \mathsf{P2}^r_{j\to k}, \mathsf{P3}^r_{j\to k} \,\}$ respectively. Since $\mathbf{Q}_{u-1} \neq \mathbf{Q}_u$, by Definition 8.1.1.3 we can say that $\mathbf{Q}_{u-1} < \mathbf{Q}_u$.
If (RULE) is (QPP_SND) or (QPP_DELIVER), then the message tag is increased and by Definition 8.1.1.3 we can conclude $\mathbf{Q}_{u-1} < \mathbf{Q}_u$.
If (RULE) is (PHS-4-SUCCESS) or (RBC-DELIVER), then a new (Lemma 8.1.2.3) message is broadcasted or an already existing message is delivered. By Definition 8.1.1.4 we have that $\mathbf{B}_{u-1} \leq \mathbf{B}_u$, and, since $\mathbf{B}_{u-1} \neq \mathbf{B}_u$, we conclude $\mathbf{B}_{u-1} < \mathbf{B}_u$. □

Finally, we give here two straightforward results on the monotonicity of message histories. The first states that the *contents* of messages that are stored within $\mathbf{Q}$ never change; only the transmission *tag* may change. The second states that round proposals never get overwritten by later updates of $\mathbf{Q}$.

Corollary 8.1.2.5 *Let $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ be a run and $X \in \{\, \mathsf{P1}^r_{j\to k}, \mathsf{P2}^r_{j\to k}, \mathsf{P3}^r_{j\to k} \,\}$, where $r \in \mathbb{N}$ and $j,k \in \mathbb{P}$. Let $u,w \in T$ with $u \leq w$.*

1. *If $\mathbf{Q}_u.X = (V,\tau)$, then there is τ' such that $\mathbf{Q}_w.X = (V,\tau')$ with $\tau \leq \tau'$.*
2. *If $\mathbf{B}_u = (v,D)$, then there is D' such that $\mathbf{B}_w = (v,D')$ with $D \supseteq D'$.*

Proof.

1. By Definition 8.1.1.3 and iterated application of Lemma 8.1.2.4.
2. By Definition 8.1.1.4 and iterated application of Lemma 8.1.2.4.

$\square$

Corollary 8.1.2.6 *Let $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ be a run. Let $\mathrm{val}^r(\mathbf{Q}_u) \neq \bot$, for some $u \in T$ and round $r > 0$, then for all $w \in T$, $w \geq u$, it holds that $\mathrm{val}^r(\mathbf{Q}_w) = \mathrm{val}^r(\mathbf{Q}_u)$.*

Proof. By the definition of $\mathrm{val}^r(\mathbf{Q})$ and Corollary 8.1.2.5 for the case 2nd-phase messages. $\square$

8.1.3 Knowledge About Reachable States

In this section we prove a number of properties of the states that can be reached by the algorithm. These results will be crucial for proving Agreement and Termination.

In any Consensus run, no program counter can be skipped. More precisely, for any given process i, if this process reaches some program counter, then it must have passed through all the earlier valid program counters as well.

Lemma 8.1.3.1 (Program counters cannot be skipped) *Let $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ be a run. Let $t \in T$ and $i \in \mathbb{P}$ such that $\mathbf{S}_t(i) = \big((r_t, P_t), \cdot, \cdot\big)$.*
Then, for all valid program counters (r,P) with $(0,\mathsf{W}) \leq_i (r,P) <_i (r_t,P_t)$, there is $u \in T$, $u < t$, such that $\mathbf{S}_u(i) = \big((r,P), \cdot, \cdot\big)$.

Proof. By induction on $t \in T$, corresponding to the length of $\langle\, \mathbf{B}_0, \mathbf{Q}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \,\rangle \rightarrow^t \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle$.

Base case $t = 0$: By definition, for all $i \in \mathbb{P}$, we have $\mathbf{S}_t(i) = \mathbf{S}_0^{(v_1,\ldots,v_n)} = \big((0,\mathsf{W}), \cdot, \cdot\big)$. By definition, there is no r and P such that $(r,P) <_i (0,\mathsf{W})$, so the statement is trivially true.

Inductive case $t > 0$: We prove the step from $t-1$ to t referring to the step

$$\langle \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \rangle \rightarrow_{j:(\text{RULE})} \langle \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \rangle \tag{8.1}$$

of R with $\mathbf{S}_{t-1}(j) = \big((r_{t-1}, P_{t-1}), \cdot, \cdot\big)$ and $\mathbf{S}_t(j) = \big((r_t, P_t), \cdot, \cdot\big)$ referring to the program counter of the process j that was touched by the last transition, where the transition rule (RULE) was applied.
By induction,

$$\begin{array}{l}\text{for all } i \in \mathbb{P} \text{ and for all valid program counters } (r, P) \text{ with} \\ (0, \mathsf{W}) \leq_i (r, P) <_i (r_{t-1}, P_{t-1}), \\ \text{there is } 0 \leq u < t-1 \text{ with } \mathbf{S}_u(i) = \big((r, P), \cdot, \cdot\big).\end{array} \tag{8.2}$$

If $i \neq j$, then the desired statement is immediately true by the induction hypothesis, because the transition from $t-1$ to t only touches the program counter of process j.
If $i = j$, then we proceed by exhaustive analysis of the cases of (RULE).

case (CRASH) This rule does not apply since $\mathbf{S}_t(j) \neq \bot$.

cases (QPP-*) and (RBC-DELIVER): With these rules, we get $(r_{t-1}, P_{t-1}) = (r_t, P_t)$, thus the statement holds immediately by the induction hypothesis.

cases (WHILE) and (PHS-*) In all these cases, we get $(r_{t-1}, P_{t-1}) <_j (r_t, P_t)$. Moreover, process i proceeds to the *immediate successor* in the program counter ordering, i.e., there is no $(\underline{r}, \underline{P})$ such that $(r_{t-1}, P_{t-1}) <_j (\underline{r}, \underline{P}) <_j (r_t, P_t)$. To conclude, we only require in addition to the induction hypothesis (8.2) that also for (r_{t-1}, P_{t-1}) there exists a $u < t$ such that $\mathbf{S}_u(i) = \big((r_{t-1}, P_{t-1}), \cdot, \cdot\big)$, which is trivially satisfied with equation (8.1) by $u = t-1$.

□

The following result states that in a consensus run messages cannot be skipped. More precisely: a process i that has reached some round r, as witnessed by its program counter in the process state, must necessarily have sent 1st- and 3rd-phase messages in all smaller rounds, plus 2nd-phase messages in the rounds smaller than r where i was coordinator.

Lemma 8.1.3.2 (Messages cannot be skipped) *Let $\big(\langle \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \rangle\big)_T^{(v_1,\ldots,v_n)}$ be a run. Let $\mathbf{S}_z(i) = ((r, P), \cdot, \cdot)$, for some $z \in T$. Let $\hat{r} \in \mathbb{N}$ with $\hat{r} \leq r$.*

1. *If $(\hat{r}, \mathsf{P1}) <_i (r, P)$, then $\mathbf{Q}_z.\mathsf{P1}^{\hat{r}}_{i\rightarrow\text{crd}(\hat{r})} \neq \bot$.*
2. *If $(\hat{r}, \mathsf{P2}) <_i (r, P)$ and $i = \text{crd}(\hat{r})$, then $\mathbf{Q}_z.\mathsf{P2}^{\hat{r}}_{\text{crd}(\hat{r})\rightarrow k} \neq \bot$ for all $k \in \mathbb{P}$.*

3. *If* $(\hat{r}, \mathsf{P3}) <_i (r, P)$*, then* $\mathbf{Q}_z.\mathsf{P3}^{\hat{r}}_{i\to\mathrm{crd}(\hat{r})} \neq \bot$.

In the following lemma we prove that the stamp contained in the current belief of a process state is always smaller than or equal to the current round number.

Lemma 8.1.3.3 (Stamp consistency in process states) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)^{(v_1,\ldots,v_n)}_T$ *be a run.*
Let $t \in T$ *and* $i \in \mathbb{P}$ *with* $\mathbf{S}_t(i) = \big((r_t, P_t), (\cdot, s_t), \cdot\big)$.

1. *If* $P_t \in \{\mathsf{P4}, \mathsf{W}\}$ *then* $s_t \leq r_t$.
2. *If* $P_t \in \{\mathsf{P1}, \mathsf{P2}, \mathsf{P3}\}$ *then* $s_t < r_t$.
3. *If* $z \in T$*, with* $t \leq z$ *and* $\mathbf{S}_z(i) = \big((r_z, P_z), (\cdot, s_z), \cdot\big)$*, then* $s_t \leq s_z$.

The previous consistency lemma for process state information directly carries over to messages.

Lemma 8.1.3.4 (Stamp consistency in process proposal) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)^{(v_1,\ldots,v_n)}_T$ *be a run. Let* $\mathbf{Q}_z.\mathsf{P1}^{r}_{i\to\mathrm{crd}(r)}{=}((\cdot, s), \cdot)$ *for some* $z \in T$*, round* $r \in \mathbb{N}$*, stamp* $s \in \mathbb{N}$ *and process* $i \in \mathbb{P}$*. Then:*

1. $s < r$
2. *for all* r' *with* $0 < r' < r$*, we have* $\mathbf{Q}_z.\mathsf{P1}^{r'}_{i\to\mathrm{crd}(r')}{=}((\cdot, s'), \cdot)$*, for some* $s' \in \mathbb{N}$ *with* $s' \leq s$.

Proof.

1. 1st-phase messages can only be created by applying rule (PHS-1), where the belief component of the current state is copied into the new 1st-phase message. As rule (PHS-1) can only be applied in phase $\mathsf{P1}$, by Lemma 8.1.3.3(2) it follows that $s < r$.
2. As $\mathbf{Q}_z.\mathsf{P1}^{r}_{i\to\mathrm{crd}(r)}{=}((\cdot, s), \cdot)$, by Lemma 8.1.2.2(1) there is $u \in T$ with $u \leq z$, for which

$$\langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle \;\rightarrow_{i:(\textsc{phs}\text{-}1)}\; \langle\, \mathbf{B}_u, \mathbf{Q}_u, \mathbf{S}_u \,\rangle$$

where $\mathbf{S}_{u-1}(i){=}\big((r, \mathsf{P1}), (\cdot, s), \cdot\big)$ and $\mathbf{S}_u(i){=}\big((r, P), (\cdot, s), \cdot\big)$, with $P = \mathsf{P2}$ if $i = \mathrm{crd}(r)$ and $P = \mathsf{P3}$ otherwise.
Let us fix an arbitrary $r' < r$. By Definition 8.1.1.1, it holds that $(r', \mathsf{P1}) <_i (r, P)$.

By Lemma 8.1.3.2(1), we get that $\mathbf{Q}_u.\mathsf{P1}^{r'}_{i\to\mathrm{crd}(r')} \neq \bot$.
Let $\mathbf{Q}_u.\mathsf{P1}^{r'}_{i\to\mathrm{crd}(r')}{=}((\cdot, s'), \cdot)$ for some $s' \in \mathbb{N}$.
Again, by Lemma 8.1.2.2(1) there is $t \in T$ for which

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\to_{i:(\text{PHS-1})} \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle$$

where $\mathbf{S}_{t-1}(i){=}\big((r', \mathsf{P1}), (\cdot, s'), \cdot\big)$ and $\mathbf{S}_t(i){=}\big((r', P), (\cdot, s'), \cdot\big)$, with $P' \in \{\mathsf{P2}, \mathsf{P3}\}$. Since $r' < r$, by Definition 8.1.1.1 we have $(r', P') <_i (r, \mathsf{P1})$; by Definition 8.1.1.2 it follows that $\mathbf{S}_t(i) < \mathbf{S}_u(i)$.
By Lemma 8.1.2.1 process states are non-decreasing, and hence $t < u$. By Lemma 8.1.3.3(3), we derive $s' \leq s$. By Corollary 8.1.2.5 and $\mathbf{Q}_u.\mathsf{P1}^{r'}_{i\to\mathrm{crd}(r')}{=}((\cdot, s'), \cdot)$, we also have $\mathbf{Q}_z.\mathsf{P1}^{r'}_{i\to\mathrm{crd}(r')}{=}((\cdot, s'), \cdot)$.

□

The next lemma states how the proposal of a process, i.e., its 1st-phase message, correlates with the 3rd-phase message that the process itself emitted in the previous round. In fact, the 3rd-phase message is an acknowledgment for the round proposal: depending on whether the process acknowledges positively or negatively, it adopts the round proposal as its own new belief or it keeps its old one. It is this possibly updated belief that the process then sends as its proposal to the coordinator of the following round.

Lemma 8.1.3.5 (Proposal creation) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)^{(v_1,\ldots,v_n)}_T$ *be a run. Let* $\mathbf{Q}_z.\mathsf{P1}^{r}_{i\to\mathrm{crd}(r)} = ((v, s), \cdot)$ *and* $\mathbf{Q}_z.\mathsf{P1}^{r+1}_{i\to\mathrm{crd}(r+1)} = ((v', s'), \cdot)$, *for some* $z \in T$, *round* $r > 0$, *values* $v, v' \in \mathbb{V}$, *and stamps* $s, s' \in \mathbb{N}$.

1. *If* $\mathbf{Q}_z.\mathsf{P3}^{r}_{i\to\mathrm{crd}(r)} = (\mathrm{nack}, \cdot)$, *then* $v = v'$ *and* $s = s'$.
2. *If* $\mathbf{Q}_z.\mathsf{P3}^{r}_{i\to\mathrm{crd}(r)} = (\mathrm{ack}, \cdot)$, *then* $v' = \mathrm{val}^r(\mathbf{Q}_z)$ *and* $s < s' = r$.

Proof.

1. Note that, since $\mathbf{Q}_z.\mathsf{P3}^{r}_{i\to\mathrm{crd}(r)} = (\mathrm{nack}, \cdot)$, rule (PHS-3-SUSPECT) tells us that $i \neq \mathrm{crd}(r)$.
   As $\mathbf{Q}_z.\mathsf{P3}^{r}_{i\to\mathrm{crd}(r)} = (\mathrm{nack}, \cdot)$, by Lemma 8.1.2.2(3a) there is $u \in T$, $u \leq z$, for which

$$\langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle \;\to_{i:(\text{PHS-3-SUSPECT})} \langle\, \mathbf{B}_u, \mathbf{Q}_u, \mathbf{S}_u \,\rangle$$

   such that

$$\begin{array}{rclcrcl} \mathbf{Q}_{u-1}.\mathsf{P3}^{r}_{i\to\mathrm{crd}(r)} & = & \bot & \quad & \mathbf{S}_{u-1}(i) & = & \big((r, \mathsf{P3}), \hat{b}, \cdot\big) \\ \mathbf{Q}_u.\mathsf{P3}^{r}_{i\to\mathrm{crd}(r)} & = & (\mathrm{nack}, \cdot) & & \mathbf{S}_u(i) & = & \big((r, \mathsf{W}), \hat{b}, \cdot\big)\,. \end{array}$$

As $\mathbf{Q}_z.\mathsf{P1}^r_{i\to\mathrm{crd}(r)} = ((v,s),\cdot)$, by Lemma 8.1.2.2(1) there is $t \in T$, $t \leq z$, for which

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\to_{i:(\text{PHS-1})} \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle$$

such that

$$\begin{array}{rclrcl} \mathbf{Q}_{t-1}.\mathsf{P1}^r_{i\to\mathrm{crd}(r)} & = & \bot & \mathbf{S}_{t-1}(i) & = & \big((r,\mathsf{P1}),(v,s),\cdot\big) \\ \mathbf{Q}_t.\mathsf{P1}^r_{i\to\mathrm{crd}(r)} & = & \big((v,s),\cdot\big) & \mathbf{S}_t(i) & = & \big((r,\mathsf{P3}),(v,s),\cdot\big)\ . \end{array}$$

As $\mathbf{Q}_z.\mathsf{P1}^{r+1}_{i\to\mathrm{crd}(r+1)} = \big((v',s'),\cdot\big)$, by Lemma 8.1.2.2(1) there is $w \in T$, $w \leq z$, for which

$$\langle\, \mathbf{B}_{w-1}, \mathbf{Q}_{w-1}, \mathbf{S}_{w-1} \,\rangle \;\to_{i:(\text{PHS-1})} \langle\, \mathbf{B}_w, \mathbf{Q}_w, \mathbf{S}_w \,\rangle$$

such that

$$\begin{array}{ll} \mathbf{Q}_{w-1}.\mathsf{P1}^{r+1}_{i\to\mathrm{crd}(r+1)} = \bot & \mathbf{S}_{w-1}(i) = \big((r{+}1,\mathsf{P1}),(v',s'),\cdot\big) \\ \mathbf{Q}_w.\mathsf{P1}^{r+1}_{i\to\mathrm{crd}(r+1)} = \big((v',s'),\cdot\big) & \mathbf{S}_w(i) = \big((r{+}1,P),(v',s'),\cdot\big)\ . \end{array}$$

where, if $i = \mathrm{crd}(r{+}1)$ then $P = \mathsf{P2}$, otherwise $P = \mathsf{P3}$.
By Lemma 8.1.2.1 process states are non-decreasing.
As $\mathbf{S}_t(i) < \mathbf{S}_u(i) < \mathbf{S}_w(i)$, it follows that $t < u < w$ and the three transitions above must be ordered in time as follows:

$$\begin{array}{lll} \langle\, \mathbf{B}_0, \mathbf{Q}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \,\rangle & \to^* & \langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \\ & \to_{i:(\text{PHS-1})} & \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle \\ & \to^* & \langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle \\ & \to_{i:(\text{PHS-3-SUSPECT})} & \langle\, \mathbf{B}_u, \mathbf{Q}_u, \mathbf{S}_u \,\rangle \\ & \to^* & \langle\, \mathbf{B}_{w-1}, \mathbf{Q}_{w-1}, \mathbf{S}_{w-1} \,\rangle \\ & \to_{i:(\text{PHS-1})} & \langle\, \mathbf{B}_w, \mathbf{Q}_w, \mathbf{S}_w \,\rangle \\ & \to^* & \langle\, \mathbf{B}_z, \mathbf{Q}_z, \mathbf{S}_z \,\rangle\ . \end{array}$$

Since $i \neq \mathrm{crd}(r)$, and since $\mathbf{S}_t(i)$ and $\mathbf{S}_{u-1}(i)$ have the same program counter $(r,\mathsf{P3})$, the only rules involving process i that can have been applied in the derivation sequence

$$\langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle \;\to^* \langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle$$

are (RBC-DELIVER), (QPP_SND), and (QPP_DELIVER). None of these rules change the current belief of process i. As a consequence, $\hat{b} = (v,s)$.

A similar reasoning applies to the derivation sequence

$$\langle\, \mathbf{B}_u, \mathbf{Q}_u, \mathbf{S}_u \,\rangle \;\rightarrow^* \langle\, \mathbf{B}_{w-1}, \mathbf{Q}_{w-1}, \mathbf{S}_{w-1} \,\rangle$$

where $(r{+}1, \mathsf{P1})$ of $\mathbf{S}_{w-1}$ is the immediate successor to (r, W) of $\mathbf{S}_u$. Thus, in the displayed sequence, rule (WHILE) *must* have been applied, while otherwise the only rules involving process i that *can* have been applied are (RBC-DELIVER), (QPP_SND), and (QPP_DELIVER). None of these rules change the current belief of process i. As a consequence, $\hat{b}{=}(v', s')$, but this implies also $(v', s'){=}(v, s)$. So, $\mathbf{Q}_w.\mathsf{P1}^{r+1}_{i\rightarrow\mathrm{crd}(r+1)}$ $= ((v, s), \cdot)$. By Corollary 8.1.2.5, we get $\mathbf{Q}_z.\mathsf{P1}^{r+1}_{i\rightarrow\mathrm{crd}(r+1)}{=}((v, s), \cdot)$, as required.

2. As $\mathbf{Q}_z.\mathsf{P3}^{r}_{i\rightarrow\mathrm{crd}(r)} = (\mathrm{ack}, \cdot)$, by Lemma 8.1.2.2(3b) there is $t \in T$, $t \leq z$, for which

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{i:(\textsc{phs-3-trust})} \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle$$

such that

$$\begin{array}{rcl@{\qquad}rcl} \mathbf{Q}_{t-1}.\mathsf{P3}^{r}_{i\rightarrow\mathrm{crd}(r)} & = & \bot & \mathbf{S}_{t-1}(i) & = & \big((r, \mathsf{P3}), \cdot, \cdot\big) \\ \mathbf{Q}_{t}.\mathsf{P3}^{r}_{i\rightarrow\mathrm{crd}(r)} & = & (\mathrm{ack}, \cdot) & \mathbf{S}_{t}(i) & = & \big((r, P), (\mathrm{val}^r(\mathbf{Q}_{t-1}), r), \cdot\big) \end{array}$$

where, if $i = \mathrm{crd}(r)$ then $P = \mathsf{P4}$, otherwise $P = \mathsf{W}$.
As $\mathbf{Q}_z.\mathsf{P1}^{r+1}_{i\rightarrow\mathrm{crd}(r+1)} = \big((v', s'), \cdot\big)$, by Lemma 8.1.2.2(1) there is $u \in T$, $u \leq z$, for which

$$\langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle \;\rightarrow_{i:(\textsc{phs-1})} \langle\, \mathbf{B}_u, \mathbf{Q}_u, \mathbf{S}_u \,\rangle$$

such that,

$$\begin{array}{rcl@{\qquad}rcl} \mathbf{Q}_{u-1}.\mathsf{P1}^{r+1}_{i\rightarrow\mathrm{crd}(r+1)} & = & \bot & \mathbf{S}_{u-1}(i) & = & \big((r{+}1, \mathsf{P1}), (v', s'), \cdot\big) \\ \mathbf{Q}_{u}.\mathsf{P1}^{r+1}_{i\rightarrow\mathrm{crd}(r+1)} & = & \big((v', s'), \cdot\big) & \mathbf{S}_{u}(i) & = & \big((r{+}1, P'), (v', s'), \cdot\big) \end{array}$$

where, if $i = \mathrm{crd}(r{+}1)$ then $P' = \mathsf{P2}$, otherwise $P' = \mathsf{P3}$.
By Lemma 8.1.2.1 process states are non-decreasing. As $\mathbf{S}_t(i) < \mathbf{S}_u(i)$, it follows that $t < u$, and the two transitions are ordered in time as follows:

$$\begin{array}{lll} \langle\, \mathbf{B}_0, \mathbf{Q}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \,\rangle & \rightarrow^* & \langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \\ & \rightarrow_{i:(\textsc{phs-3-trust})} & \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle \\ & \rightarrow^* & \langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle \\ & \rightarrow_{i:(\textsc{phs-1})} & \langle\, \mathbf{B}_u, \mathbf{Q}_u, \mathbf{S}_u \,\rangle \\ & \rightarrow^* & \langle\, \mathbf{B}_z, \mathbf{Q}_z, \mathbf{S}_z \,\rangle \end{array}$$

By Lemma 8.1.2.1 process states are non-decreasing. As a consequence, the only rules involving process i that can have been applied in the derivation sequence $\langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle \rightarrow^* \langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle$ and that can possibly have affected its state are: (PHS-4-FAIL), (PHS-4-SUCCESS), (WHILE) and (RBC-DELIVER). However, none of these rules affects the belief component of the state, and hence $(v', s') = (\mathrm{val}^r(\mathbf{Q}_{t-1}), r)$. By Corollary 8.1.2.6 $\mathrm{val}^r(\mathbf{Q}_{t-1}) = \mathrm{val}^r(\mathbf{Q}_z)$. As a consequence, $v' = \mathrm{val}^r(\mathbf{Q}_z)$ and $s' = r$. Finally, since $\mathbf{Q}_z.\mathsf{P1}^r_{i\to\mathrm{crd}(r)} = ((v,s),\cdot)$, by Lemma 8.1.3.4(1) it follows that $s < r$.

□

We need a last technical lemma in order to show how process proposals are generated.

Lemma 8.1.3.6 *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run. Let* $z \in T$, $r, r' \in \mathbb{N}$ *with* $0 < r' < r$, *and* $i \in \mathbb{P}$ *such that* $\mathbf{Q}_z.\mathsf{P1}^{r'}_{i\to\mathrm{crd}(r')} = ((v',s'),\cdot)$ *and* $\mathbf{Q}_z.\mathsf{P1}^{r}_{i\to\mathrm{crd}(r)} = ((v,s),\cdot)$.

1. *If* $s' = s$ *then* $v' = v$ *and for all* r'', $r' \le r'' < r$, *it holds that* $\mathbf{Q}_z.\mathsf{P3}^{r''}_{i\to\mathrm{crd}(r'')} = (\mathrm{nack},\cdot)$.
2. *If* $s' < s$ *then there is a round* $\hat{r}$, $r' \le \hat{r} < r$, *such that* $\mathbf{Q}_z.\mathsf{P3}^{\hat{r}}_{i\to\mathrm{crd}(\hat{r})} = (\mathrm{ack},\cdot)$.

The following lemma states that the proposal (v,s) proposed by some process i in a certain round r originates precisely from round s.

Lemma 8.1.3.7 (Proposal origin) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run and* $z \in T$. *If* $\mathbf{Q}_z.\mathsf{P1}^{r}_{i\to\mathrm{crd}(r)} = ((v,s),\cdot)$, *with* $r > 1$ *and* $s > 0$, *then* $\mathbf{Q}_z.\mathsf{P3}^{s}_{i\to\mathrm{crd}(s)} = (\mathrm{ack},\cdot)$ *and* $v = \mathrm{val}^s(\mathbf{Q}_z)$.

Proof. By $\mathbf{Q}_z.\mathsf{P1}^{r}_{i\to\mathrm{crd}(r)} = ((v,s),\cdot)$ and Lemma 8.1.3.4(1) it follows that $s < r$. By Lemma 8.1.3.4(2) and $s < r$, we derive $\mathbf{Q}_z.\mathsf{P1}^{s}_{i\to\mathrm{crd}(s)} \neq \bot$ and $\mathbf{Q}_z.\mathsf{P1}^{s+1}_{i\to\mathrm{crd}(s+1)} \neq \bot$ (the case $s{+}1 = r$ is trivial). Let $\mathbf{Q}_z.\mathsf{P1}^{s}_{i\to\mathrm{crd}(s)} = ((v_s,r_s),\cdot)$, by Lemma 8.1.3.4(1) we have $r_s < s$.

Now, suppose by contradiction that $\mathbf{Q}_z.\mathsf{P3}^{s}_{i\to\mathrm{crd}(s)} = (\mathrm{nack},\cdot)$. Then, with Lemma 8.1.3.5(1), also $\mathbf{Q}_z.\mathsf{P1}^{s+1}_{i\to\mathrm{crd}(s+1)} = ((v_s,r_s),\cdot)$. As $r_s < s$, and by (matching $s{+}1/r_s$ with r'/s' of) Lemma 8.1.3.6(2), there is some $\hat{r}$ with $s{+}1 \le \hat{r} < r$ (and hence $s < \hat{r}$) such that $\mathbf{Q}_z.\mathsf{P3}^{\hat{r}}_{i\to\mathrm{crd}(\hat{r})} = (\mathrm{ack},\cdot)$. By an application of Lemma 8.1.3.5(2), we get $\mathbf{Q}_z.\mathsf{P1}^{\hat{r}+1}_{i\to\mathrm{crd}(\hat{r}+1)} = ((\mathrm{val}^{\hat{r}}(\mathbf{Q}_z),\hat{r}),\cdot)$. As $\mathbf{Q}_z.\mathsf{P1}^{r}_{i\to\mathrm{crd}(r)} = ((v,s),\cdot)$, by an application Lemma 8.1.3.4(2) we get the contradiction $\hat{r} \le s$. As a consequence, it must be $\mathbf{Q}_z.\mathsf{P3}^{s}_{i\to\mathrm{crd}(s)} = (\mathrm{ack},\cdot)$.

By Lemma 8.1.3.5(2) and $\mathbf{Q}_z.\mathsf{P3}^s_{i\to\mathrm{crd}(s)} = (\mathrm{ack}, \cdot)$, we get $\mathbf{Q}_z.\mathsf{P1}^{s+1}_{i\to\mathrm{crd}(s+1)} = ((\mathrm{val}^s(\mathbf{Q}_z), s), \cdot)$. This together with Lemma 8.1.3.6(1) allows us to derive $v = \mathrm{val}^s(\mathbf{Q}_z)$.

□

The next lemma states that if the coordinator broadcasts a value in a certain round, then it must have previously—in the very same round—received a majority of (positive) acknowledgments on its round proposal.

Lemma 8.1.3.8 (Coordinator's RBC-broadcast) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)^{(v_1,\ldots,v_n)}_T$ *be a run, and* $z \in T$. *If* $\mathbf{B}_z(r){\neq}\bot$ *then* $|\,\mathrm{ack}(\mathrm{received}(\mathbf{Q}_z).\mathsf{P3}^r)| \geq \lceil (n{+}1)/2 \rceil$.

Proof. If $\mathbf{B}(r){\neq}\bot$ by Lemma 8.1.2.2(4) there is $t \in T$, for which

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \ \to_{\mathrm{crd}(r):(\textsc{phs-4-success})} \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle$$

such that, for some $v \in \mathbb{V}$

$$\begin{array}{rclrcl} \mathbf{B}_{t-1}(r) & = & \bot & \mathbf{S}_{t-1}(\mathrm{crd}(r)) & = & \big((r,\mathsf{P4}), (v,\cdot), \cdot\big) \\ \mathbf{B}_t(r) & = & (v, \mathbb{P}) & \mathbf{S}_t(\mathrm{crd}(r)) & = & \big((r,\mathsf{W}), (v,\cdot), \cdot\big) \,. \end{array}$$

The premises of rule (PHS-4-SUCCESS) require

$$|\,\mathrm{ack}(\mathrm{received}(\mathbf{Q}_{t-1}).\mathsf{P3}^r)| \geq \lceil (n{+}1)/2 \rceil.$$

Again, by Lemma 8.1.2.4, message histories are non-decreasing and hence $\mathbf{Q}_{t-1} \leq \mathbf{Q}_t \leq \mathbf{Q}_z$. As a consequence,

$$\begin{array}{rcl} |\,\mathrm{ack}(\mathrm{received}(\mathbf{Q}_z).\mathsf{P3}^r)| & \geq & |\,\mathrm{ack}(\mathrm{received}(\mathbf{Q}_t).\mathsf{P3}^r)| \\ & \geq & |\,\mathrm{ack}(\mathrm{received}(\mathbf{Q}_{t-1}).\mathsf{P3}^r)| \\ & \geq & \lceil (n{+}1)/2 \rceil \,. \end{array}$$

□

8.2 Consensus Properties

In this section, we prove the three main Consensus properties: *Validity*, *Agreement*, and *Termination*.

Apart from Validity (Section 8.2.1), in our opinion, it is not obvious how to turn the intuitive, but rather informal, correctness arguments of §3.1.2 into rigorous proofs. Most of the arguments given there use the concept of round, which is orthogonal to the concept of time in a run. With our proof of Agreement (Section 8.2.2) and its supporting set of lemmas, we demonstrate how to carefully recover the events belonging to some round from the details of a given run. The high-level proof structure is very similar to Chandra and Toueg's proof in [12]. For Termination (§8.2.3), however, we needed to come up with different proof ideas and structures, mainly because our underlying notion of run is different from the one in [12].

8.2.1 Validity

Intuitively, in a generic run, whenever a value appears as an estimate, a proposal, or a decision —i.e., within one of the components $\mathbf{B}$, $\mathbf{Q}$ or $\mathbf{S}$— then this value is one of the initially proposed values. In other words, values are never invented. The Validity property of the Consensus algorithm coincides with the last item of the following theorem.

Theorem 8.2.1.1 (Validity) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run. Then, for all* $t \in T$*, the conjunction of the following properties holds:*

1. *If* $\mathbf{B}_t(r) = (v, \cdot)$*, for some* $0 < r$*, then there is* $k \in \mathbb{P}$ *with* $v = v_k$*.*
2. *If* $\mathbf{Q}_t.\mathsf{P1}^r_{i\to\mathrm{crd}(r)} = ((v,\cdot),\cdot)$*, for some* $i \in \mathbb{P}$ *and* $0 < r$*, then there is* $k \in \mathbb{P}$ *with* $v = v_k$*.*
3. *If* $\mathbf{Q}_t.\mathsf{P2}^r_{\mathrm{crd}(r)\to i} = ((v,\cdot),\cdot)$*, for some* $i \in \mathbb{P}$ *and* $0 < r$*, then there is* $k \in \mathbb{P}$ *with* $v = v_k$*.*
4. *If* $\mathbf{S}_t(i) = \big(\cdot, (v,\cdot), \cdot\big)$*, for some* $i \in \mathbb{P}$*, then there is* $k \in \mathbb{P}$ *with* $v = v_k$*.*
5. *If* $\mathbf{S}_t(i) = \big(\cdot, \cdot, (v,\cdot)\big)$*, for some* $i \in \mathbb{P}$*, then there is* $k \in \mathbb{P}$ *with* $v = v_k$*.*

Proof. By induction on the length t of the (prefix of a) run

$$\langle\, \mathbf{B}_0, \mathbf{Q}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \,\rangle \;\rightarrow^t\; \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle.$$

Base case $t = 0$ **:** Immediate since $\mathbf{B}_0 = \emptyset$, $\mathbf{Q}_0 = \emptyset$ and $\mathbf{S}(i) = \mathbf{S}_0^{(v_1,\ldots,v_n)} = \big((0,\mathsf{W}), (v_i, 0), \bot\big)$ for $i \in \mathbb{P}$.

Inductive case $t > 0$ **:** We prove the step from $t-1$ to t (if $t \in T$) referring to the step

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{j:(\textsc{Rule})}\; \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle \tag{8.3}$$

By exhaustive analysis, we check for every rule (RULE) that each of the five conditions is satisfied in configuration $\langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle$. All cases are similar: in every possible step, values that appear in the conclusions of the applied rule are just copied via the premises referring to the former configuration, such that the inductive hypothesis applies and proves the required origin in the initial configuration.

case $(\textsc{Rule}) = (\textsc{phs-2})$ **for some** $0 < r$**:**

1. $\mathbf{B}_t = \mathbf{B}_{t-1}$, so the inductive hypothesis is already sufficient.
2. $\mathbf{Q}_t.\mathsf{P1}^r = \mathbf{Q}_{t-1}.\mathsf{P1}^r$ for all $0 < r$, so the inductive hypothesis is already sufficient.
3. $\mathbf{Q}_t = \mathbf{Q}_{t-1}\{\,\mathsf{P2}^r_{\mathrm{crd}(r)\to j} \mapsto (\mathrm{best}^r(\mathbf{Q}_{t-1}), \mathrm{outgoing})\,\}$, so in addition to the inductive hypothesis on $\mathbf{Q}_{t-1}$, we only need to show that $\mathrm{best}^r(\mathbf{Q}_{t-1}) = v_k$ for some $k \in \mathbb{P}$. This is guaranteed by the projective definition of $\mathrm{best}^r(\cdot)$ together with the inductive hypothesis on $\mathbf{Q}_{t-1}$.
4. Since $\mathbf{S}_{t-1}(j) = ((r, \mathsf{P2}), b, d)$ and $\mathbf{S}_t(j) = ((r, \mathsf{P3}), b, d)$ mention the same value in b, and since $\mathbf{S}_t(i) = \mathbf{S}_{t-1}(i)$ for all $i \neq j$, the inductive hypothesis is already sufficient.
5. Completely analogous to the previous case, except arguing with d instead of b.

case $(\textsc{Rule}) \neq (\textsc{phs-2})$**:** Analogous to the previous, just that there is not even an indirection referring to an externally defined function like $\mathrm{best}^r(\cdot)$.

□

8.2.2 Agreement

In the Introduction we said that a value gets *locked* as soon as enough processes have, in the same round, positively acknowledged the estimate proposed by the coordinator of that round (the round proposal). In our terminology, a value v is *locked* for round r (in a message history $\mathbf{Q}$) if enough positive 3rd-phase messages are defined for round r; the transmission state of these ack-messages is not important, be it $\mathrm{outgoing}$, $\mathrm{transit}$, or $\mathrm{received}$.

Definition 8.2.2.1 *A value v is called* locked *for round r in a* Qpp*-message history $\mathbf{Q}$, written $\mathbf{Q} \overset{r}{\looparrowright} v$, if $v = \mathrm{val}^r(\mathbf{Q})$ and $|\,\mathrm{ack}(\mathbf{Q}.\mathsf{P3}^r)| \geq \lceil (n{+}1)/2 \rceil$.*

Notice the convenience of having at hand the history abstraction to access the messages that were sent in the past, without having to look at the run leading to the current state. This is independent of possible crashes of the senders of the messages.

We now show that whenever a Rbc-broadcast occurs, it is for a locked value.

Lemma 8.2.2.2 (Locking) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run where* $\mathbf{B}_z(r) = (v,\cdot)$ *for some* $z \in T$, $0 < r \in \mathbb{N}$, $v \in \mathbb{V}$. *Then* $\mathbf{Q}_z \overset{r}{\looparrowright} v$.

Proof. Since $\mathbf{B}_0(r) = \bot$ and $\mathbf{B}_z(r) \neq \bot$, by Lemma 8.1.2.2(4), there exists $t \in T$, with $t \leq z$, for which

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \ \rightarrow_{\mathrm{crd}(r):(\textsc{phs-4-success})} \langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle$$

such that, for some $\hat{v} \in \mathbb{V}$ and $\hat{r} \in \mathbb{N}$,

$$\begin{array}{rclrcl} \mathbf{B}_{t-1}(r) & = & \bot & \mathbf{S}_{t-1}(\mathrm{crd}(r)) & = & \big((r,\mathsf{P4}),(\hat{v},\hat{r}),\cdot\big) \\ \mathbf{B}_t(r) & = & (\hat{v},\mathbb{P}) & \mathbf{S}_t(\mathrm{crd}(r)) & = & \big((r,\mathsf{W}),(\hat{v},\hat{r}),\cdot\big)\ . \end{array}$$

As $(r,\mathsf{P3}) <_{\mathrm{crd}(r)} (r,\mathsf{P4})$, by Lemma 8.1.3.2(3), we have $\mathbf{Q}_t.\mathsf{P3}^r_{\mathrm{crd}(r)\rightarrow\mathrm{crd}(r)} \neq \bot$. As the coordinator cannot suspect itself, it must be $\mathbf{Q}_t.\mathsf{P3}^r_{\mathrm{crd}(r)\rightarrow\mathrm{crd}(r)} = (\mathrm{ack},\cdot)$. Thus, by an application of Lemma 8.1.2.2(3b), there exists $u \in T$, $u < t$ (notice that it cannot be $u = t$), for which

$$\langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle \ \rightarrow_{\mathrm{crd}(r):(\textsc{phs-3-trust})} \langle\, \mathbf{B}_u, \mathbf{Q}_u, \mathbf{S}_u \,\rangle$$

such that

$$\mathbf{S}_{u-1}(\mathrm{crd}(r)) = \big((r,\mathsf{P3}),\cdot,\cdot\big) \text{ and } \quad \mathbf{S}_u(\mathrm{crd}(r)) = \big((r,\mathsf{P4}),(\mathrm{val}^r(\mathbf{Q}_{u-1}),r),\cdot\big).$$

Since $\mathbf{S}_u(\mathrm{crd}(r))$ and $\mathbf{S}_{t-1}(\mathrm{crd}(r))$ have the same program counter $(r,\mathsf{P4})$, the only rules involving $\mathrm{crd}(r)$ that can have been applied in the derivation sequence $\langle\, \mathbf{B}_u, \mathbf{Q}_u, \mathbf{S}_u \,\rangle \rightarrow^* \langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle$ are (QPP_SND), (QPP_DELIVER) and (RBC-DELIVER). However, rules (QPP_SND) and (QPP_DELIVER) do not affect the state of process $\mathrm{crd}(r)$, while (RBC-DELIVER) affects only the decision component while keeping the belief component unchanged. As a consequence, $\hat{v} = \mathrm{val}^r(\mathbf{Q}_{u-1})$ and $\hat{r} = r$.

As $\mathbf{B}_z(r) = (v,\cdot)$, $\mathbf{B}_t(r) = (\hat{v},\cdot)$ and $t \leq z$, by Corollary 8.1.2.5(2), we then have $v = \hat{v} = \mathrm{val}^r(\mathbf{Q}_{u-1})$. By Corollary 8.1.2.6, we obtain

$$v = \mathrm{val}^r(\mathbf{Q}_z)\ .$$

Moreover, since $\mathbf{B}_z(r) \neq \bot$, by Lemma 8.1.3.8 we have

$$|\,\mathrm{ack}(\mathbf{Q}_z).\mathsf{P3}^r| \geq |\,\mathrm{ack}(\mathrm{received}(\mathbf{Q}_z).\mathsf{P3}^r)| \geq \lceil (n{+}1)/2 \rceil\ .$$

From these two results, by applying Definition 8.2.2.1, we derive $\mathbf{Q}_z \overset{r}{\looparrowright} v$. □

Now, we can move to look at agreement properties. The key idea is to prove that lockings in two different rounds must be for the very same value.

Proposition 8.2.2.3 (Locking agreement) *Let $\big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ be a run. If there are a state $z \in T$, round numbers $r_1, r_2 \in \mathbb{N}$ and values $v_1, v_2 \in \mathbb{V}$ such that $\mathbf{Q}_z \overset{r_1}{\looparrowright} v_1$ and $\mathbf{Q}_z \overset{r_2}{\looparrowright} v_2$, then $v_1 = v_2$.*

Proof. Let us assume $r_1 \leq r_2$ (the proof for the opposite case is analogous). From Definition 8.2.2.1, we have $v_1 = \mathrm{val}^{r_1}(\mathbf{Q}_z)$ and $v_2 = \mathrm{val}^{r_2}(\mathbf{Q}_z)$. We will prove that $\mathrm{val}^{r_1}(\mathbf{Q}_z) = \mathrm{val}^{r_2}(\mathbf{Q}_z)$ by strong induction on r_2 starting from r_1.

Base case If $r_1 = r_2$ then the result is trivially true.
Inductive case Let $r_2 > r_1$.

By inductive hypothesis we assume that for all $r \in \mathbb{N}$, $r_1 \leq r \leq r_2{-}1$,

$$\mathrm{val}^{r}(\mathbf{Q}_z) = \mathrm{val}^{r_1}(\mathbf{Q}_z) = v_1 .$$

By definition of $\mathrm{val}^{r_2}(\mathbf{Q}_z)$ (Definition 6.2.0.2), we have $\mathbf{Q}_z.\mathsf{P2}^{r_2}_{\mathrm{crd}(r_2)\to i} = ((\mathrm{val}^{r_2}(\mathbf{Q}_z), \cdot), \cdot)$, for all $i \in \mathbb{P}$. By Lemma 8.1.2.2(2), there is $t \in T$, $t \leq z$, for which

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \ \to_{\mathrm{crd}(r_2):(\textsc{phs}\text{-}2)} \langle\, \mathbf{B}_{t}, \mathbf{Q}_{t}, \mathbf{S}_{t} \,\rangle$$

such that, for some $b \in \mathbb{V} \times \mathbb{N}$ and for all $i \in \mathbb{P}$

$$\mathbf{Q}_{t-1}.\mathsf{P2}^{r_2}_{\mathrm{crd}(r_2)\to i} = \bot \qquad \mathbf{S}_{t-1}(\mathrm{crd}(r_2)) = \big((r_2, \mathsf{P2}), b, \cdot\big)$$

$$\mathbf{Q}_{t}.\mathsf{P2}^{r_2}_{\mathrm{crd}(r_2)\to i} = (\mathrm{best}^{r_2}(\mathbf{Q}_{t-1}), \mathrm{outgoing})$$

$$\mathbf{S}_{t}(\mathrm{crd}(r_2)) = \big((r_2, \mathsf{P3}), b, \cdot\big) \ .$$

From Table 6.2, we see that one of the premises of rule ($\textsc{phs}$-2) is $|\,\mathrm{received}(\mathbf{Q}_{t-1}).\mathsf{P1}^{r_2}| \geq \lceil (n{+}1)/2 \rceil$.
Let $C_{r_2} = \{\, j \mid \mathbf{Q}_z.\mathsf{P1}^{r_2}_{j\to \mathrm{crd}(r_2)} = (\cdot, \mathrm{received}) \,\}$, then by Corollary 8.1.2.5(1) on the monotonicity on message histories we have

$$|\, C_{r_2} \,| \geq |\, \mathrm{received}(\mathbf{Q}_{t-1}).\mathsf{P1}^{r_2} \,| \geq \lceil (n{+}1)/2 \rceil .$$

Let $L_{r_1} = \{\, j \mid \mathbf{Q}_z.\mathsf{P3}^{r_1}_{j\to \mathrm{crd}(r_1)} = (\mathrm{ack}, \cdot) \,\}$. As $\mathbf{Q}_z \overset{r_1}{\looparrowright} v_1$, by Definition 8.2.2.1 we have $|\,\mathrm{ack}(\mathbf{Q}_z.\mathsf{P3}^{r_1})| \geq \lceil (n{+}1)/2 \rceil$ and hence $|\, L_{r_1} \,| \geq \lceil (n{+}1)/2 \rceil$.
Since the cardinality of both C_{r_2} and L_{r_1} is greater than $\lceil (n{+}1)/2 \rceil$, and $C_{r_2}, L_{r_1} \subseteq \{1, \ldots, n\}$, it follows that $L_{r_1} \cap C_{r_2} \neq \emptyset$. Let us take a process $k \in L_{r_1} \cap C_{r_2}$.
As $k \in C_{r_2}$, we have $\mathbf{Q}_z.\mathsf{P1}^{r_2}_{k\to \mathrm{crd}(r_2)} \neq \bot$. As $r_2 \geq r_1{+}1$, by Lemma 8.1.3.4(2) also $\mathbf{Q}_z.\mathsf{P1}^{r_1+1}_{k\to \mathrm{crd}(r_1+1)} \neq \bot$.

Since $k \in L_{r_1}$, we have $\mathsf{Q}_z.\mathsf{P3}^{r_1}_{k\to\mathrm{crd}(r_1)} = (\mathrm{ack}, \cdot)$. By Lemma 8.1.3.5(2)

$$\mathsf{Q}_z.\mathsf{P1}^{r_1+1}_{k\to\mathrm{crd}(r_1+1)} = \big(\big(\,\mathrm{val}^{r_1}(\mathbf{Q}_z), r_1\big), \cdot\big)\ .$$

Since $k \in C_{r_2}$, we have

$$\mathsf{Q}_z.\mathsf{P1}^{r_2}_{k\to\mathrm{crd}(r_2)} = \big((\cdot, \tilde{r}), \mathrm{received}\,\big)$$

for some $\tilde{r} \in \mathbb{N}$. As $r{+}1 \leq r_2$, by Lemma 8.1.3.4(2) we have $r_1 \leq \tilde{r}$. Let us now consider the process $i \in C_{r_2}$ that has proposed the best value in round r_2, according to Definition 6.2.0.1. Then by Definition 6.2.0.2 we have

$$\mathsf{Q}_z.\mathsf{P1}^{r_2}_{i\to\mathrm{crd}(r_2)} = \big(\big(\,\mathrm{val}^{r_2}(\mathbf{Q}_z), \hat{r}\big), \mathrm{received}\,\big)$$

for some $\hat{r} \in \mathbb{N}$. By applying Lemma 8.1.3.4(1), we get $\hat{r} < r_2$. As i has proposed in round r_2 the best value, with the best timestamp $\hat{r}$, it follows that $\tilde{r} \leq \hat{r}$.
Putting it altogether, we get $r_1 \leq \tilde{r} \leq \hat{r} \leq r_2{-}1$. By the inductive hypothesis, $\mathrm{val}^{\hat{r}}(\mathbf{Q}_z) = \mathrm{val}^{r_1}(\mathbf{Q}_z)$.
From $\mathsf{Q}_z.\mathsf{P1}^{r_2}_{i\to\mathrm{crd}(r_2)} = \big(\big(\,\mathrm{val}^{r_2}(\mathbf{Q}_z), \hat{r}\big), \mathrm{received}\,\big)$ and Lemma 8.1.3.7, we get that $\mathrm{val}^{\hat{r}}(\mathbf{Q}_z) = \mathrm{val}^{r_2}(\mathbf{Q}_z)$. Thus, $\mathrm{val}^{r_1}(\mathbf{Q}_z) = \mathrm{val}^{r_2}(\mathbf{Q}_z)$, as required.

□

Now, everything is in place to prove the property of Agreement: Whenever two processes i and j have decided, as expressed within the respective state components, they have done so for the same value.

Theorem 8.2.2.4 (Agreement) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x\,\rangle\big)^{(v_1,\ldots,v_n)}_T$ *be a run. If there are a time* $z \in T$*, processes* $i, j \in \mathbb{P}$*, and values* $v_1, v_2 \in \mathbb{V}$ *such that* $\mathbf{S}_z(i) = (\cdot, \cdot, (v_1, \cdot))$ *and* $\mathbf{S}_z(j) = (\cdot, \cdot, (v_2, \cdot))$*, then* $v_1 = v_2$*.*

Proof. In the initial state $\mathbf{S}_0$ the decision component of both process i and process j is undefined (see Section 6.1), while in state $\mathbf{S}_z$ it is defined. By inspection of the rules of Table 6.2, the decision element of a process state can be changed only once per run, by applying rule (RBC-DELIVER). In particular, it can change from $\perp$ to something of the form $(\hat{v}, \hat{r})$ executing rule (RBC-DELIVER) for the first time. When executing rule (RBC-DELIVER) the following times, the decision value is left unchanged. Therefore, once initialized, the decision value of a process state stays the same until the end of the run. As a consequence, there are $t \in T$ and $u \in T$, with $t \leq z$ and $u \leq z$, such that

$$\langle\, \mathbf{B}_{t-1}, \mathbf{Q}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{i:(\textsc{Rbc-deliver})}\; \langle\, \mathbf{B}_{t}, \mathbf{Q}_{t}, \mathbf{S}_{t} \,\rangle$$

with $\mathbf{S}_{t-1}(i) = (\cdot,\cdot,\bot)$, $\mathbf{S}_t(i) = (\cdot,\cdot,(v_1,r_1))$, $\mathbf{B}_{t-1}(r_1) = (v_1,\cdot)$ for some $r_1 > 0$, and

$$\langle\, \mathbf{B}_{u-1}, \mathbf{Q}_{u-1}, \mathbf{S}_{u-1} \,\rangle \;\rightarrow_{j:(\textsc{Rbc-deliver})}\; \langle\, \mathbf{B}_{u}, \mathbf{Q}_{u}, \mathbf{S}_{u} \,\rangle$$

with $\mathbf{S}_{u-1}(j) = (\cdot,\cdot,\bot)$, $\mathbf{S}_u(i) = (\cdot,\cdot,(v_2,r_2))$, $\mathbf{B}_{u-1}(r_2) = (v_2,\cdot)$ for some $r_2 > 0$.

As $u, t \leq z$, by Corollary 8.1.2.5(2), $\mathbf{B}_z(r_1) = (v_1,\cdot)$ and $\mathbf{B}_z(r_2) = (v_2,\cdot)$.

By Lemma 8.2.2.2, it follows that $\mathbf{Q}_z \overset{r_1}{\looparrowright} v_1$ and $\mathbf{Q}_z \overset{r_2}{\looparrowright} v_2$.

By Proposition 8.2.2.3, we conclude $v_1 = v_2$.

□

Recall that the *Agreement* property states that two correct processes cannot decide different values. There is actually also a stronger property, called *Uniform Agreement*, requiring that any two processes (no matter whether correct or faulty) cannot decide different values. Since in the proof of Theorem 8.2.2.4 the correctness of processes is not required, the Chandra and Toueg's algorithm actually satisfies the property of *Uniform Agreement* [12].

8.2.3 Termination

The layout and conduct of the proof of Termination is strongly influenced by the fundamental definition of runs and their required properties. A standard approach in the field of Distributed Algorithms is that of considering system runs of infinite length. This is also the path followed by Chandra and Toueg [12]: they work under the important fairness assumption that ”every correct process takes an infinite number of steps”. On the other hand, when working with transition systems, as we do here, it makes sense to admit complete runs of finite length (cf. §4). We find it more adequate and intuitive to capture the termination of a single Distributed Consensus by means of a run that takes only finitely many steps. In fact, for complete finite runs, a particular fairness property holds automatically: since no transition is enabled in the final configuration of a complete finite run, any previously (permanently) enabled transition must have been executed in this run.

To compare the different layouts of Chandra and Toueg's proof of Termination and ours, we first briefly sketch how the former works. The critical part of [12, Lemma 6.2.2 (Termination)] is to capture the notion of *blocking of processes* when waiting for messages or circumstances that might never arrive. This argument is formulated as ”*no correct process remains blocked forever at one of the* **wait** *statements*”. The associated reasoning (by contradiction) starts with the identification, in a given run, of a process that is blocked in some program counter; should there be several processes that are blocked in the run, the one with the smallest program counter is chosen. As we understand it, a process is blocked in a given run if there exists a configuration after which the process stays forever in the same program counter. Thus, blocking can only occur if (A) no transition is ever enabled, or –note that we consider infinite runs, aka: linear-time behaviors– if (B) an enabled transition is never executed. Due to the above-mentioned fairness assumption on “correct process progress” in infinite runs and the fact that processes are essentially single-threaded, every enabled transition would eventually be executed, so case B is immediately excluded. Then, under carefully chosen assumptions of the contradiction setup one can show that transitions would always eventually become enabled, so also case A is excluded. For this reasoning, the main ingredients are (1) the fairness assumptions on QPP- and RBC-delivery, (2) the FD-properties of $\Diamond\mathcal{S}$ and (3) the assumption on a majority of correct processes.

The decisive role of the definition of infinite runs, and its accompanying fairness assumption, in Chandra and Toueg’s proof is obvious. Not surprisingly, since our definition of run is different in that it admits finite runs and comes without explicit fairness assumptions, also our Termination proof is different. Moreover, our proof layout is conveniently structured in that it allows us to separate concerns. First, we prove as an independent result that admissible runs are always finite. Then, the bare analysis of the last configuration of an admissible run allows us to reason about the blocking of processes in a very simple way; no fairness assumptions need to be considered, because no transitions are enabled anyway.

We start proving that all admissible runs of Consensus (Definition 6.4.5.1) are finite. The proof relies on the next lemma showing that in a hypothetic infinite run of Consensus every round $r \in \mathbb{N}$ is reached by at least one process. Intuitively, this holds since the (finite) set of processes cannot stay within their respective rounds for an infinite number of computation steps. Unless all processes decide or crash, at least one of them must proceed to higher rounds. The reader may observe some similarity to the ”correct process progress” fairness assumption in infinite runs of Chandra and Toueg.

Lemma 8.2.3.1 (Infinity) *Let* $\left(\langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle\right)_{\mathbb{N}}^{(v_1,\ldots,v_n)}$ *be an infinite run of Con-*

sensus. Then, for all rounds $r > 0$, there are $i \in \mathbb{P}$ and $t \in \mathbb{N}$ such that $\mathbf{S}_t(i) = \big((r, \mathsf{P3}), \cdot, \cdot\big)$.

Proof. By contradiction. Assume there is $r > 0$ such that, for all $i \in \mathbb{P}$ and $t \in \mathbb{N}$, it holds that $\mathbf{S}_t(i) \neq \big((r, \mathsf{P3}), \cdot, \cdot\big)$. By applying Lemma 8.1.3.1 we have:

$$\text{for all } i \in \mathbb{P} \text{ and } t \in \mathbb{N}, \text{ if } \mathbf{S}_t(i) = \big((r_t, P_t), \cdot, \cdot\big), \text{ then } (r_t, P_t) <_i (r, \mathsf{P3}). \tag{8.4}$$

Note that, for every process i, the number of program counters (r_t, P_t) smaller than $(r, \mathsf{P3})$ is finite, and since there are only five phases, the number of the rounds that can be reached by any process i is finite too.

In the following, we show that the number of reductions that a process may perform at a certain program counter (r_t, P_t) is finite. For this, we compute an upper bound on the maximum number of possible derivations that a process may perform at program counter (r_t, P_t). We proceed by case analysis on the transition rules for an arbitrary $i \in \mathbb{P}$:

1. The rules (WHILE), (PHS-1), (PHS-2), (PHS-3-TRUST), (PHS-3-SUSPECT), (PHS-4-FAIL), (PHS-4-SUCCESS), and (CRASH) cause the change of the program counter. Thus, by Lemma 8.1.2.1, each of them can only be performed once for each program counter.
2. All remaining rules do not change the program counter, so they are critical for the desired bound.
   (a) The rule (RBC-DELIVER) can be performed for each occurrence of some $\mathbf{B}(\hat{r}) = (v, D)$ such that D contains i. However, every execution of (RBC-DELIVER) removes i from the respective D, which corresponds to the property of RBC-Integrity of Definition 6.4.3.1, such that for each $\hat{r}$, process i can execute (RBC-DELIVER) only once. How many possible $\hat{r}$ are there? Entries $\mathbf{B}(\hat{r})$ can only be produced by the rule (PHS-4-SUCCESS), once per round number $\hat{r}$ that has been reached by the corresponding coordinator. Since we have shown above with (8.4) that only finitely many round numbers are reached, there are only finitely many possible applications of (RBC-DELIVER) for process i.
   (b) The rule (QPP_SND) can be performed at most $r_t{*}(n{+}2)$ times; where $n{+}2$ is the maximum number of point-to-point messages that a process may produce in a round. We multiply this number for r_t assuming the worst case that none of the messages produced in the previous rounds has yet left the site.
   (c) The rule (QPP_DELIVER) can be performed at most $r_t{*}(2{*}n + 1)$ times. In fact, the message addressed to a certain process in a

given round are at most $2{*}n{+}1$, when the process is coordinator of the round. We multiply this number for r_t supposing that none of the messages sent in the previous rounds has yet been received by the site.

Summing up, since there are only finitely many possible program counters (r_t, P_t) smaller than $(r, \mathsf{P3})$, and since each process can only perform a finite number of steps while staying at such a counter, the whole run can only contain a finite number of computation steps. This concludes the contradiction. □

Now, everything is in place to prove that all the admissible runs of Consensus are finite. This is done by contradiction, assuming that there is an infinite run of Consensus. In this case, the presence of FD Ω and the constraint that only a minority of processes can crash ensure that, at a certain round, some coordinator broadcasts its decision. The properties of Reliable Broadcast are then used to show that the run should be finite, thus obtaining a contradiction.

Theorem 8.2.3.2 (Finiteness of Admissible Runs) *All admissible runs of Consensus are finite.*

Proof. We proceed by contradiction. Assume there is an *infinite* admissible run of Consensus

$$R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_{\mathbb{N}}^{(v_1,\ldots,v_n)} .$$

As an auxiliary definition, let $\mathrm{max\_rnd}(\mathbf{Q}_x) := \max\{\, r \mid (r, \cdot, \cdot, \cdot, \cdot, \cdot) \in \mathbf{Q}_x \,\}$ denote the greatest round number for which, by time x, any QPP-message has occurred.

As incorrect processes are doomed to crash, there is a time $t_{\text{crashed}} \in \mathbb{N}$ such that for all $u \geq t_{\text{crashed}}$ and $j \in \mathbb{P} \setminus \mathsf{Correct}(R)$ it holds that $\mathbf{S}_u(j) = \bot$. As R is admissible, the Ω-property (Definition 6.4.4.1) ensures that there is a time $t_{\text{trusted}} \in \mathbb{N}$ after which some process $i \in \mathsf{Correct}(R)$ is never again suspected. Let $t = \max\{t_{\text{trusted}}, t_{\text{crashed}}\}$. Let r be the smallest round such that $r > \mathrm{max\_rnd}(\mathbf{Q}_t)$ and $i = \mathrm{crd}(r)$. By this construction, as process i is no longer suspected after time t, process i is not suspected in R in any round greater or equal than r. Also, if $(r, \cdot, \cdot, \cdot, \cdot, \cdot) \in \mathbf{Q}_u$, at some time u, then $t < u$.

In the following, we first derive that, in R, process i passes through round r and that it executes rule (PHS-4-SUCCESS) while doing so. Using this knowledge, in a second step, we use the RBC-properties of the admissible run R to derive a contradiction.

First, let us move to round $r{+}n{+}1$. As R is infinite, by Lemma 8.2.3.1 there must exist a time $z \in \mathbb{N}$, such that for some process $k \in \mathbb{P}$ we have $\mathbf{S}_z(k) = \big((r{+}n{+}1, \mathsf{P3}), \cdot, \cdot\big)$.

Now, we move backwards to round $r{+}n$. As $(r{+}n, \mathsf{P3}) <_k (r{+}n{+}1, \mathsf{P3})$, by Lemma 8.1.3.2(3) and $i = \mathrm{crd}(r) = \mathrm{crd}(r{+}n)$, we get $\mathbf{Q}_z.\mathsf{P3}^{r+n}_{k\to i} \neq \bot$. By Lemma 8.1.2.2(3), in round $r + n$, process k must have executed either rule (PHS-3-SUSPECT) or rule (PHS-3-TRUST). However, as process i is not suspected in any round beyond r according to Definition 6.4.4.1, it is rule (PHS-3-TRUST) that process k must have executed in round $r + n$. Thus, with Lemma 8.1.2.2(3b) we have $\mathbf{Q}_z.\mathsf{P3}^{r+n}_{k\to i} = (\mathrm{ack}, \cdot)$. Moreover, the premises of (PHS-3-TRUST) require the presence of the 2nd-phase message sent by the coordinator i to process k. Formally, by Corollary 8.1.2.5(1), $\mathbf{Q}_z.\mathsf{P2}^{r+n}_{i\to k} \neq \bot$. By Lemma 8.1.2.2(2), this implies that in round $r + n$ process i executed rule (PHS-2) in state $\mathbf{S}_w(i) = \big((r + n, \mathsf{P2}), \cdot, \cdot\big)$ for some $w \in \mathbb{N}$. (Intuitively, this means that the coordinator i must itself have reached round $r{+}n$.)

Now, we move back to round r. As $i = \mathrm{crd}(r)$ and $(r, \mathsf{P4}) <_i (r, \mathsf{W}) <_i (r{+}n, \mathsf{P2})$, by Lemma 8.1.3.1 and Lemma 8.1.2.1 there is a time $u < w$ such that $\mathbf{S}_{u-1}(i) = \big((r, \mathsf{P4}), \cdot, \cdot\big)$ and $\mathbf{S}_u(i) = \big((r, \mathsf{W}), \cdot, \cdot\big)$. By inspection of the transition rules of Tables 6.1 and 6.2, the state $\mathbf{S}_u(i)$ can only be reached by executing either rule (PHS-4-SUCCESS) or rule (PHS-4-FAIL). The premises of both rules require that in round r process i must have received a majority of 3rd-phase messages from the other processes. Formally, $|\,\mathrm{received}(\mathbf{Q}_{u-1}).\mathsf{P3}^r| \geq \lceil (n{+}1)/2 \rceil$. As process i is not suspected in round r, it follows that i must have received in round r only *positive* 3rd-phase messages. Formally, $|\,\mathrm{ack}(\mathrm{received}(\mathbf{Q}_{u-1}).\mathsf{P3}^r)| \geq \lceil (n{+}1)/2 \rceil$. As a consequence, process i in round r must have reached the state $\mathbf{S}_u(i) = \big((r, \mathsf{W}), \cdot, \cdot\big)$ by firing rule (PHS-4-SUCCESS).

By the definition of rule (PHS-4-SUCCESS) and Lemma 8.1.2.3(4), we have $\mathbf{B}_{u-1}(r) = \bot$ and $\mathbf{B}_u(r) = (v, \mathbb{P})$ for some $v \in \mathbb{V}$ and $r \in \mathbb{N}$. This means that in round r process i has RBC-broadcast the value v. As R is admissible (Definition 6.4.5.1), the *Validity* RBC*-property* (Definition 6.4.3.1(1)) guarantees that, in R, process i eventually RBC-delivers the value v by executing rule (RBC-DELIVER). Likewise, the *Agreement* RBC*-property* (Definition 6.4.3.1(2)) guarantees that, in R, all processes in $\mathsf{Correct}(R)$ eventually RBC-deliver v and decide by executing rule (RBC-DELIVER). Formally, with Definition 6.4.3.1(2) and rule (RBC-DELIVER), there is a time $t_{\mathrm{delivered}}$ such that for each $j \in \mathsf{Correct}(R)$ there exists a value $d_j \neq \bot$ such that $\mathbf{S}_{t_{\mathrm{delivered}}}(j) = \big((\cdot, \cdot), \cdot, d_j\big)$. Notice that rule (RBC-DELIVER) is the only one that acts on the decision component of process states; moreover, once this component is defined its value cannot change afterwards.

Now, we are ready to derive the contradiction. Observe that, by construction (recall that R is infinite), we have $t_{\text{crashed}} \leq t < u < t_{\text{delivered}}$. Let us consider some round $\hat{r} > 1 + \text{max\_rnd}(\mathbf{Q}_{t_{\text{delivered}}})$. As R is infinite, by Lemma 8.2.3.1, there are $k \in \mathbb{P}$ and $z \in \mathbb{N}$ such that $\mathbf{S}_z(k) = \big((\hat{r}, \mathsf{P3}), \cdot, \cdot\big)$. Since $\hat{r}$ is only reached well beyond time t_{crashed}, we have that $k \in \mathsf{Correct}(R)$. Since program counters cannot be skipped, by Lemma 8.1.3.1, there must be a time where k passed through both previous subsequent counters $(\hat{r}{-}1, \mathsf{W}) <_k (\hat{r}, \mathsf{P1})$. The only way to do this is by executing rule (WHILE) at round $\hat{r}{-}1$, but this is not possible since according to our construction $d_k \neq \bot$ must have been already defined at round $\hat{r} - 1 > \text{max\_rnd}(\mathbf{Q}_{t_{\text{delivered}}})$. Contradiction.

□

In what follows, we prove that all correct processes of an admissible Consensus run eventually decide in that run. Some terminology will be helpful to discuss this theorem.

Definition 8.2.3.3 (Process termination and deadlock) *Let*

$$\langle\, \mathbf{B}_0, \mathbf{Q}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \,\rangle \;\rightarrow^* \langle\, \mathbf{B}, \mathbf{Q}, \mathbf{S} \,\rangle \not\rightarrow$$

be a finite run of Consensus. With respect to the final state of this run, a process $i \in \mathbb{P}$ is called

1. crashed *if* $\mathbf{S}(i) = \bot$*;*
2. deadlocked *if* $\mathbf{S}(i) = \big(\cdot, \cdot, \bot\big)$*;*
3. terminated *if* $\mathbf{S}(i) = \big(\cdot, \cdot, d\big)$*, with* $d \neq \bot$*.*

Note that both *deadlocked* and *terminated* processes represent *blocked* behaviors, but due to the condition $d \neq \bot$ the latter is considered successful, while the former is not. Since the three cases of the Definition 8.2.3.3 cover all possibilities for the possible states of $\mathbf{S}(i)$, it is obvious that in the final state, all processes are either crashed, deadlocked, or terminated.

Since we know by Theorem 8.2.3.2 that admissible runs are finite, we may conveniently focus our analysis of process termination on the last configuration of a given admissible run. To support this kind of reasoning, the following lemma describes the precise conditions under which processes can be deadlocked in such a final configuration, depending on their role in the algorithm at this last moment.

Lemma 8.2.3.4 (Process deadlock) *Let* $R = \big(\langle\, \mathbf{B}_t, \mathbf{Q}_t, \mathbf{S}_t \,\rangle\big)^{(v_1,\ldots,v_n)}_{[0,\ldots,z]}$ *be an admissible run of Consensus. Let $i \in \mathbb{P}$ be deadlocked, that is $\mathbf{S}_z(i) = ((r, P), \cdot, \bot)$, for some r and P.*

1. *If* $i \neq \mathrm{crd}(r)$ *then* $P = \mathsf{P3}$, $\mathbf{S}_z(\mathrm{crd}(r)) \neq \bot$, *and* $\mathbf{Q}_z.\mathsf{P2}^r_{\mathrm{crd}(r)\to i} = \bot$*;*
2. *If* $i = \mathrm{crd}(r)$ *then*
   (a) either $P = \mathsf{P2}$ *and* $|\,\mathrm{received}(\mathbf{Q}_z).\mathsf{P1}^r\,| < \lceil (n{+}1)/2 \rceil$
   (b) or $P = \mathsf{P4}$ *and* $|\,\mathrm{received}(\mathbf{Q}_z).\mathsf{P3}^r\,| < \lceil (n{+}1)/2 \rceil$.

Proof. As R is complete, but finite, we have that $\langle\, \mathbf{B}_z, \mathbf{Q}_z, \mathbf{S}_z \,\rangle \not\rightarrow$.

1. Let $i \neq \mathrm{crd}(r)$. We proceed by case analysis on the possible phases $P \in \{\, \mathsf{P1}, \mathsf{P3}, \mathsf{W} \,\}$ of non-coordinating participants.

   If $P = \mathsf{P1}$ then rule (PHS-1) could be applied without further condition. As $\langle\, \mathbf{B}_z, \mathbf{Q}_z, \mathbf{S}_z \,\rangle \not\rightarrow$, it follows that $P \neq \mathsf{P1}$.
   If $P = \mathsf{W}$ then rule (WHILE) could be applied since $\mathbf{S}_z(i)) = \big((r, P), \cdot, \bot\big)$. As $\langle\, \mathbf{B}_z, \mathbf{Q}_z, \mathbf{S}_z \,\rangle \not\rightarrow$, it follows that $P \neq \mathsf{W}$.
   If $P = \mathsf{P3}$ then two rules might apply: (PHS-3-TRUST) and (PHS-3-SUSPECT).
   As $\langle\, \mathbf{B}_z, \mathbf{Q}_z, \mathbf{S}_z \,\rangle \not\rightarrow$ both rules must not meet the conditions to fire. More precisely, since rule (PHS-3-TRUST) cannot apply, then it must be $\mathbf{Q}_z.\mathsf{P2}^r_{\mathrm{crd}(r)\to i} \neq (\cdot, \mathrm{received})$. By definition of QPP-reliability (Definition 6.4.2.1) this implies $\mathbf{Q}_z.\mathsf{P2}^r_{\mathrm{crd}(r)\to i} = \bot$. Moreover, since rule (PHS-3-SUSPECT) cannot apply then $\mathrm{crd}(r)$ has not crashed, i.e. $\mathbf{S}_z(\mathrm{crd}(r)) \neq \bot$.

2. Let $i = \mathrm{crd}(r)$. Again, we proceed by case analysis on $P \in \{$ P1, P2, P3, P4, W $\}$. The cases $P = \mathsf{P1}$ and $P = \mathsf{W}$ are identical to the previous case of $i \neq \mathrm{crd}(r)$, since the respective rules are independent of i being $\mathrm{crd}(r)$, or not.

   case P2 The only possibility to prevent an application of rule (PHS-2) is by *not* satisfying $|\,\mathrm{received}(\mathbf{Q}_z).\mathsf{P1}^r\,| \geq \lceil (n{+}1)/2 \rceil$, so we have $|\,\mathrm{received}(\mathbf{Q}_z).\mathsf{P1}^r\,| < \lceil (n{+}1)/2 \rceil$.
   case P3 This case is different from the respective one for $i \neq \mathrm{crd}(r)$ that we have seen above: with $i = \mathrm{crd}(r)$ only the rule (PHS-3-TRUST) *may* be applicable. And we show here that it actually *is*. By Lemma 8.1.3.2(2), since $(r, \mathsf{P2}) <_i (r, \mathsf{P3})$ and $i = \mathrm{crd}(r)$, we derive $\mathbf{Q}_z.\mathsf{P2}^r_{\mathrm{crd}(r)\to i} \neq \bot$. As $\langle\, \mathbf{B}_z, \mathbf{Q}_z, \mathbf{S}_z \,\rangle \not\rightarrow$ and $\mathbf{S}_z(i) \neq \bot$ it follows that $i \in \mathsf{Correct}(R)$. By Definition 6.4.2.1, it follows that $\mathbf{Q}_z.\mathsf{P2}^r_{\mathrm{crd}(r)\to i} = (\cdot, \mathrm{received})$. Thus, rule (PHS-3-TRUST) applies to the final state $\mathbf{S}_z(i) = ((r, \mathsf{P3}), \cdot, \cdot)$. As $\langle\, \mathbf{B}_z, \mathbf{Q}_z, \mathbf{S}_z \,\rangle \not\rightarrow$, it follows that $P \neq \mathsf{P3}$.
   case P4 Potentially, either of the rules (PHS-4-SUCCESS) or (PHS-4-FAIL) may apply. The only possibility to prevent an application

of rule (PHS-4-FAIL) is by *not* satisfying $|\,\mathrm{received}(\mathbf{Q}_z).\mathsf{P3}^r\,| \geq \lceil(n{+}1)/2\rceil$, thus we have $|\,\mathrm{received}(\mathbf{Q}_z).\mathsf{P3}^r\,| < \lceil(n{+}1)/2\rceil$. As

$$|\,\mathrm{ack}(\mathrm{received}(\mathbf{Q}_z).\mathsf{P3}^r)\,| \leq |\,\mathrm{received}(\mathbf{Q}_z).\mathsf{P3}^r\,|$$

the condition $|\,\mathrm{received}(\mathbf{Q}_z).\mathsf{P3}^r\,| < \lceil(n{+}1)/2\rceil$ is sufficient to prevent also an application of rule (PHS-4-SUCCESS), which requires $|\,\mathrm{ack}(\mathrm{received}(\mathbf{Q}_z).\mathsf{P3}^r)\,| \geq \lceil(n{+}1)/2\rceil$.

□

We can now prove the Termination property. Again, we reason by contradiction, assuming that there exists an admissible Consensus run R in which one of the correct processes does not decide. The Agreement RBC-property is then used to derive that, in this case, no correct process decides in R. As no process decides and as admissible runs are finite, it follows that all correct processes should be deadlocked in R. This leads us to the required contradiction.

Theorem 8.2.3.5 (Termination) *Let $R = \big(\langle\,\mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x\,\rangle\big)_T^{(v_1,\ldots,v_n)}$ be an admissible run of Consensus.*
If $i \in \mathsf{Correct}(R)$, then there is a time $t \in T$ and a value d such that $\mathbf{S}_t(i) = (\cdot,\cdot,d)$, with $d \neq \bot$.

Proof. By Theorem 8.2.3.2, we have $R = \big(\langle\,\mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x\,\rangle\big)_{[0,\ldots,z]}^{(v_1,\ldots,v_n)}$, for some $z \in \mathbb{N}$, with $\langle\,\mathbf{B}_z, \mathbf{Q}_z, \mathbf{S}_z\,\rangle \not\rightarrow$.

By contradiction, assume there is a process $i \in \mathsf{Correct}(R)$ that does not decide in R. Formally, we assume that, for all $t \in T$, we have $\mathbf{S}_t(i) = (\cdot,\cdot,\bot)$. In particular, we have $\mathbf{S}_z(i) = (\cdot,\cdot,\bot)$.

By inspection of the transition rules in Tables 6.1 and 6.2, rule (RBC-DELIVER) is the only one acting on (i.e., defining) the decision component of process states; moreover, once this component is defined, its value remains unchanged. As a consequence, in R, process i cannot have performed rule (RBC-DELIVER). As R is admissible, the *Agreement* RBC*-property* (Definition 6.4.3.1(2)) guarantees that no process in $\mathsf{Correct}(R)$ RBC-delivered, and hence decided, in R. As a consequence, for all $k \in \mathsf{Correct}(R)$, we have $\mathbf{S}_z(k) = ((r_k, P_k),\cdot,\bot)$, i.e., all processes in $\mathsf{Correct}(R)$ must be deadlocked in R.

Let $\underline{k} \in \mathsf{Correct}(R)$ be the process with the smallest program counter in $\mathsf{Correct}(R)$, formally $(r_{\underline{k}}, P_{\underline{k}}) \leq (r_k, P_k)$, for all $k \in \mathsf{Correct}(R)$. Note that, since $\mathsf{Correct}(R) \neq \emptyset$, such a $\underline{k}$ always exists, although it is not uniquely defined; we simply choose one among possibly several candidates.

By Lemma 8.2.3.4, there are three possible cases for process $\underline{k}$ being deadlocked at time z:

case $\underline{k} \neq \mathrm{crd}(r_{\underline{k}})$ **and** $P_{\underline{k}} = \textsf{P3}$. By Lemma 8.2.3.4, we also have $\mathbf{S}_z(\mathrm{crd}(r_{\underline{k}})) \neq \bot$ and $\mathbf{Q}_z.\textsf{P2}^{r_{\underline{k}}}_{\mathrm{crd}(r_{\underline{k}})\to k} = \bot$, for all $k \in \textsf{Correct}(R)$. As $\mathbf{S}_z(\mathrm{crd}(r_{\underline{k}})) \neq \bot$, it follows that $\mathrm{crd}(r_{\underline{k}}) \in \textsf{Correct}(R)$.
As $\mathbf{Q}_z.\textsf{P2}^{r_{\underline{k}}}_{\mathrm{crd}(r_{\underline{k}})\to k} = \bot$, for all $k \in \textsf{Correct}(R)$, by Lemma 8.1.3.2(2), the program counter of process $\mathrm{crd}(r_{\underline{k}})$ in $\mathbf{S}_z$ must be smaller or equal than $(r_{\underline{k}}, \textsf{P2})$. However, this is in contradiction with the fact that $\underline{k}$ has the smallest program counter (which is equal to $(r_{\underline{k}}, \textsf{P3})$) among the processes in $\textsf{Correct}(R)$.

case $\underline{k} = \mathrm{crd}(r_{\underline{k}})$ **and** $P_{\underline{k}} = \textsf{P2}$. By Lemma 8.2.3.4, we also have $|\,\mathrm{received}(\mathbf{Q}_z).\textsf{P1}^{r_{\underline{k}}}\,| < \lceil (n{+}1)/2 \rceil$. Moreover, due to the minimality of $\underline{k}$, we have $(r_{\underline{k}}, \textsf{P1}) < (r_{\underline{k}}, \textsf{P2}) = (r_{\underline{k}}, P_{\underline{k}}) \leq (r_k, P_k)$, for all $k \in \textsf{Correct}(R)$. By Lemma 8.1.3.2(1), it follows that $\mathbf{Q}_z.\textsf{P1}^{r_{\underline{k}}}_{k\to\mathrm{crd}(r_{\underline{k}})} \neq \bot$, for all $k \in \textsf{Correct}(R)$. As R is admissible and $\underline{k} \in \textsf{Correct}(R)$, and by QPP-reliability (Definition 6.4.2.1), it follows that $\mathbf{Q}_z.\textsf{P1}^{r_{\underline{k}}}_{k\to\mathrm{crd}(r_{\underline{k}})} = (\cdot, \mathrm{received})$, for all $k \in \textsf{Correct}(R)$.
Thus, $|\,\mathrm{received}(\mathbf{Q}_z).\textsf{P1}^{r_{\underline{k}}}\,| = |\,\textsf{Correct}(R)\,| \geq \lceil (n{+}1)/2 \rceil$, in contradiction to what we derived at the beginning of this case.

case $\underline{k} = \mathrm{crd}(r_{\underline{k}})$ **and** $P_{\underline{k}} = \textsf{P4}$. By Lemma 8.2.3.4, we also have $|\,\mathrm{received}(\mathbf{Q}_z).\textsf{P3}^{r}\,| < \lceil (n{+}1)/2 \rceil$. Moreover, due to minimality of $\underline{k}$, we have $(r_{\underline{k}}, \textsf{P3}) < (r_{\underline{k}}, \textsf{P4}) = (r_{\underline{k}}, P_{\underline{k}}) \leq (r_k, P_k)$, for all $k \in \textsf{Correct}(R)$. By Lemma 8.1.3.2(3), it follows that $\mathbf{Q}_z.\textsf{P3}^{r_{\underline{k}}}_{k\to\mathrm{crd}(r_{\underline{k}})} \neq \bot$, for all $k \in \textsf{Correct}(R)$. As R is admissible and $\underline{k} \in \textsf{Correct}(R)$, and by QPP-reliability (Definition 6.4.2.1), it follows that $\mathbf{Q}_z.\textsf{P3}^{r_{\underline{k}}}_{k\to\mathrm{crd}(r_{\underline{k}})} = (\cdot, \mathrm{received})$ for all $k \in \textsf{Correct}(R)$.
Thus, $|\,\mathrm{received}(\mathbf{Q}_z).\textsf{P3}^{r_{\underline{k}}}\,| = |\,\textsf{Correct}(R)\,| \geq \lceil (n{+}1)/2 \rceil$, in contradiction to what we derived at the beginning of this case.

These three cases allow us to conclude that there is no process i with $\mathbf{S}_z(i) = (\cdot, \cdot, \bot)$, i.e., deadlocked in the last configuration of R. Thus, for all $k \in \textsf{Correct}(R)$, it must be $\mathbf{S}_z(k) = (\cdot, \cdot, d_k)$, with $d_k \neq \bot$. □

Chapter 9

Correctness Proofs for the Paxos Algorithm

9.1 Basic Properties of the Algorithm

In this section we prove a number of basic properties of the Consensus algorithm defined in §7. Most of them are simple, but still very useful properties holding for (prefixes of) runs of the form:

$$\langle\, \mathbf{B}_0, \mathbf{C}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \,\rangle \;\rightarrow^* \langle\, \mathbf{B}, \mathbf{C}, \mathbf{S} \,\rangle \;.$$

When carrying out proofs more informally, on the basis of pseudo code, the properties of this section are those that would typically be introduced via the words ”by inspection of the code”. To express them formally, we first need to define some appropriate orderings on the components of configurations.

9.1.1 Orderings

Natural orders can be defined on process states as well as point-to-point and broadcast histories.

Leader and phase identifiers are transitively ordered as $(\text{yes}, \text{idle}) < (\text{yes}, \text{trying}) < (\text{yes}, \text{waiting}) < (\text{yes}, \text{polling}) < (\text{yes}, \text{done}) < (\text{no}, \text{idle})$; we write $\leq$ to denote the reflexive closure of $<$. From the order on phases it is easy to derive a partial order on program counters that takes into account whether the respective process, say i, is leader of a particular round r.

For the purpose of ordering, we group together the state components $a = (l, Ph)$, r and p, we call this triplet *program counter* and we refer to it as c.

Definition 9.1.1.1 (Ordering Program Counters) *Let* $c = (a, r, p)$ *and* $c' = (a', r', p') \in \big(\{\,\text{yes}, \text{no}\,\} \times \{\,\text{idle}, \text{trying}, \text{waiting}, \text{polling}, \text{done}\,\}\big) \times \mathbb{N} \times \mathbb{N}$.

1. *We write* $c \leq c'$ *if* $p < p'$ *or* $(p = p' \wedge r < r')$ *or* $(p = p' \wedge r = r' \wedge a \leq a')$.
2. *The triplet* (a, r, p) *is a* valid program counter *for some process* i *if:*
   (a) $l = \text{no}$ *implies* $Ph = \text{idle}$,
   (b) $r = 0$ *implies* $a = (\text{no}, \text{idle})$, *and*
   (c) $r \in \mathbb{R}(i)$.
3. *We write* $c \leq_i c'$ *if* $c \leq c'$ *and both* c *and* c' *are valid for process* i.

We write $c <_i c'$ *(respectively,* $c < c'$*) if* $c \leq_i c'$ *(respectively,* $c \leq c'$*) and* $c \neq c'$.

Notice that in the definition of $\leq_i$ the validity conditions are necessary as only leaders go through phases trying,waiting and polling, as the consensus algorithm starts in (no, idle) for all the processes and as each process has a reserved set of request numbers. Due to the latter, there is no c such that $c <_i ((\text{no}, \text{idle}), 0, 0)$.

If we consider the partial order on program counters, together with the incarnation number, the information on the processes' crash status (where defined is smaller than undefined, if the process has the same incarnation number) and the value of the decision field (where undefined is smaller than defined), then we can establish a partial order on both the global state array and the local states of the processes.

Definition 9.1.1.2 (Ordering States) *Let* $\mathbf{S}$ *and* $\mathbf{S}'$ *be process states and* $i \in \mathbb{P}$. *Let* $c = (a, r, p)$ *and* $c' = (a', r', p')$, *let* $\mathbf{S}(i)=(c, (\cdot, s), (u, \iota), d)$ *and* $\mathbf{S}'(i)=(c', (\cdot, s'), (u', \iota'), d')$ *We write* $\mathbf{S}(i) \leq \mathbf{S}'(i)$ *if*

1. *either* $c <_i c'$,
2. *or* $c{=}c' \;\wedge\; \iota{<}\iota'$,
3. *or* $c{=}c' \;\wedge\; \iota{=}\iota' \;\wedge\; u{<}u'$,
4. *or* $c{=}c' \;\wedge\; \iota{=}\iota' \;\wedge\; u{=}u' \;\wedge\; s{<}s'$,
5. *or* $c{=}c' \;\wedge\; \iota{=}\iota' \;\wedge\; u{=}u' \;\wedge\; s{=}s' \;\wedge\; ((d = d') \vee d = \bot)$.

We write $\mathbf{S} \leq \mathbf{S}'$ *if for all* $i \in \mathbb{P}$ *it holds that* $\mathbf{S}(i) \leq \mathbf{S}'(i)$. *We write* $\mathbf{S}(i) < \mathbf{S}'(i)$ *if* $\mathbf{S}(i) \leq \mathbf{S}'(i)$ *and* $\mathbf{S}(i) \neq \mathbf{S}'(i)$. *Similarly, we write* $\mathbf{S} < \mathbf{S}'$ *if* $\mathbf{S} \leq \mathbf{S}'$ *and* $\mathbf{S} \neq \mathbf{S}'$.

Note that the global ordering on state arrays properly reflects the local character of the transition rules of §7.3. As we will show with Lemma 9.1.2.1 each computation step of the algorithm corresponds precisely to the state change of a single process.

Tags are ordered as follows: a message in the outgoing buffer can leave it or a message that is already in transit can be received or lost, or a message that is in transit can be duplicated.

Definition 9.1.1.3 (Ordering Tags) *Let $\tau = (\alpha, \beta, \gamma, \delta)$ and $\tau' = (\alpha', \beta', \gamma', \delta')$ $\in \mathbb{N} \times \mathbb{N} \times \mathbb{N} \times \mathbb{N}$ be two message tags. We write $\tau \leq \tau'$ if*

1. *either $\alpha' < \alpha$,*
2. *or $\alpha{=}\alpha' \;\wedge\; \delta{<}\delta'$,*
3. *or $\alpha{=}\alpha' \;\wedge\; \delta{=}\delta' \;\wedge\; \gamma{<}\gamma'$,*
4. *or $\alpha{=}\alpha' \;\wedge\; \delta{=}\delta' \;\wedge\; \gamma{=}\gamma' \;\wedge\; \beta{<}\beta'$,*

Message histories are ordered as follows: an entry defined in the later history must either be undefined in the older history, or its tag in the older history is smaller according to the tag order of Definition 9.1.1.3.

Definition 9.1.1.4 (Ordering CH-Message Histories) *Let $\mathbf{C}, \mathbf{C}'$ be CH-message histories.*
Then $\mathbf{C} \leq \mathbf{C}'$ iff for all $X \in \{\, \mathsf{P0}^r_{\mathrm{lead}(r)\to i}, \mathsf{P1}^r_{i\to\mathrm{lead}(r)}, \mathsf{P2}^r_{\mathrm{lead}(r)\to i}, \mathsf{P3}^r_{i\to\mathrm{lead}(r)} \,\}$, with $i \in \mathbb{P}$ and $r \in \mathbb{N}$:

1. *either $\mathbf{C}.X = \perp$*
2. *or $\mathbf{C}.X = (V, \tau) \;\wedge\; \mathbf{C}'.X = (V, \tau') \;\wedge\; \tau \leq \tau'$.*

As usual, we define the strict counterpart as $\mathbf{C} < \mathbf{C}'$ iff $\mathbf{C} \leq \mathbf{C}'$ and $\mathbf{C} \neq \mathbf{C}'$.

In fact, our semantics may overwrite transmission tags, but only in increasing order while leaving the other data unchanged (see Corollary 9.1.2.5). Notice that the order of CH-message histories is transitive.

Finally, broadcast histories are ordered according to the decreasing delivery set of their entries.

Definition 9.1.1.5 (Ordering RBC-Message Histories) *Let $\mathbf{B}, \mathbf{B}'$ be RBC-message histories and $r \in \mathbb{N}$. We write $\mathbf{B}(r) \leq \mathbf{B}'(r)$ iff*

1. *either $\mathbf{B}(r) = \perp$*
2. *or $\mathbf{B}(r) = (v, D)$ and $\mathbf{B}'(r) = (v, D')$ with $D \supseteq D'$.*

We write $\mathbf{B} \leq \mathbf{B}'$ if for all $r \in \mathbb{N}$ it holds that $\mathbf{B}(r) \leq \mathbf{B}'(r)$. We write $\mathbf{B}(r) < \mathbf{B}'(r)$ if $\mathbf{B}(r) \leq \mathbf{B}'(r)$ and $\mathbf{B}(r) \neq \mathbf{B}'(r)$. Similarly, we write $\mathbf{B} < \mathbf{B}'$ if $\mathbf{B} \leq \mathbf{B}'$ and $\mathbf{B} \neq \mathbf{B}'$.

As for CH-message histories, overwriting of RBC-message histories may happen, but only decreasing the delivery set, while keeping the recorded broadcast value unchanged (see Lemma 9.1.2.4).

9.1.2 Monotonicity and Message Evolution

In every Consensus run, states and message histories are monotonically non-decreasing. We start from the monotonicity of the state component, which is derived from the respective program counters.

Lemma 9.1.2.1 (Monotonicity of States) *Let $R = \big(\langle \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \rangle\big)_T^{(v_1,\ldots,v_n)}$ be a run. Let*

$$\langle \mathbf{B}_{z-1}, \mathbf{C}_{z-1}, \mathbf{S}_{z-1} \rangle \rightarrow_{i:(\text{RULE})} \langle \mathbf{B}_z, \mathbf{C}_z, \mathbf{S}_z \rangle$$

be an arbitrary computation step of R, for some $z \in T$, $i \in \mathbb{P}$, with $\mathbf{S}_{z-1}(i) = (c_{z-1}, (\cdot, s_{z-1}), (u_{z-1}, \iota_{z-1}), d_{z-1})$ and $\mathbf{S}_z(i) = (c_z, (\cdot, s_z), (u_z, \iota_z), d_z)$.

1. *If* (RULE) *is rule* (CRASH), *then $c_{z-1} = c_z$, $\iota_{z-1} = \iota_z$ and $u_{z-1} = \top < \bot = u_z$; hence $\mathbf{S}_{z-1}(i) < \mathbf{S}_z(i)$ and $\mathbf{S}_{z-1} < \mathbf{S}_z$.*
2. *If* (RULE) *is rule* (RECOVERY), *then $c_{z-1} \le c_z$ and $\iota_{z-1} < \mathrm{succ}(\iota_{z-1}) = \iota_z$; hence $\mathbf{S}_{z-1}(i) < \mathbf{S}_z(i)$ and $\mathbf{S}_{z-1} < \mathbf{S}_z$.*
3. *If* (RULE) *is any of the rules* (CH_SND) *or* (CH_DELIVER) *or* (CH_DUPL) *or* (CH_LOSS), *then $c_{z-1} = c_z$, $\iota_{z-1} = \iota_z$, $u_{z-1} = u_z$, and $d_{z-1} = d_z$; hence $\mathbf{S}_{z-1}(i) = \mathbf{S}_z(i)$, and $\mathbf{S}_{z-1} = \mathbf{S}_z$.*
4. *If* (RULE) *is any of the rules* (LEADER-YES) *or* (LEADER-NO), *then $c_{z-1} < c_z$, and hence $\mathbf{S}_{z-1}(i) < \mathbf{S}_z(i)$, and $\mathbf{S}_{z-1} < \mathbf{S}_z$.*
5. *If* (RULE) *is any of the rules* (NEW), (SND-PREP-REQ), (RCV-PREP-REQ), (SND-ACC-REQ), (SUCCESS) *then $c_{z-1} < c_z$, and hence $\mathbf{S}_{z-1}(i) < \mathbf{S}_z(i)$, and $\mathbf{S}_{z-1} < \mathbf{S}_z$.*
6. *If* (RULE) *is rule* (RCV-ACC-REQ), *$c_{z-1} = c_z$, $\iota_{z-1} = \iota_z$, $u_{z-1} = u_z$, $s_{z-1} < s_z$ hence $\mathbf{S}_{z-1}(i) < \mathbf{S}_z(i)$, and $\mathbf{S}_{z-1} < \mathbf{S}_z$.*
7. *If* (RULE) *is rule* (RBC-DELIVER), *$c_{z-1} = c_z$, $\iota_{z-1} = \iota_z$, $u_{z-1} = u_z$, $s_{z-1} = s_z$ and $d_{z-1} = d_z$ or $d_{z-1} = \bot$; hence $\mathbf{S}_{z-1}(i) \le \mathbf{S}_z(i)$, and $\mathbf{S}_{z-1} \le \mathbf{S}_z$.*

Proof. Immediate from the local comparison of premises and conclusions of each individual rule (RULE), and Definition 9.1.1.2 of the state ordering derived from program counters. □

Proving the monotonicity of the CH-message history $\mathbf{C}$ and of the RBC-message history $\mathbf{B}$ takes instead a bit more effort (see Lemma 9.1.2.4). Analyzing the transition rules in Table 7.1 and 7.2, we see that there are precisely four rules that *add* messages to a component $\mathbf{C}$, namely (SND-PREP-REQ), (RCV-PREP-REQ), (SND-ACC-REQ), and (RCV-ACC-REQ). Moreover, there are four rules that change messages in a component $\mathbf{C}$, namely (CH_SND), (CH_DELIVER), (CH_DUPL) and (CH_LOSS). All the other rules leave the component $\mathbf{C}$ untouched. As for $\mathbf{B}$, there is only one rule that adds messages to it: (SUCCESS). Moreover, the application of rule (RBC-DELIVER) changes the content of messages in a component $\mathbf{B}$ while all other rules leave $\mathbf{B}$ untouched.

The following lemma describes how P0, P1, P2, P3 and broadcast messages are created, essentially by reading the transition rules backwards and identifying appropriate conditions. More precisely, from the definedness of an entry in the CH-message or in the RBC-message history of some reachable configuration, we can derive that there must have been along the way (starting from the initial configuration) a uniquely defined computation step, where this particular entry was first added. This addition must have been produced by one of the aforementioned five rules. In order to uniquely identify such steps, we decompose the run that led to the assumed reachable configuration into five parts, as follows.

Lemma 9.1.2.2 (Message Creation) *Let $R = \big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ be a run. Then,*

1. *If $\mathbf{C}_u.\mathsf{P0}^r_{\mathrm{lead}(r)\to k} \neq \bot$, for some $u \in T$, $k \in \mathbb{P}$, and $r > 0$, then there is $t \in T$, $t \leq u$, for which*

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{\mathrm{lead}(r):(\text{SND-PREP-REQ})} \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle$$

   such that, for all $i \in \mathbb{P}$,

$$\mathbf{C}_{t-1}.\mathsf{P0}^r_{\mathrm{lead}(r)\to i} = \bot \qquad \mathbf{S}_{t-1}(\mathrm{lead}(r)) = \big((\mathrm{yes}, \mathrm{trying}), r, \cdot, \cdot, \cdot, \cdot\big)$$

$$\mathbf{C}_t.\mathsf{P0}^r_{\mathrm{lead}(r)\to i} = (r, (\iota, 0, 0, 0)) \quad \mathbf{S}_t(\mathrm{lead}(r)) = \big((\mathrm{yes}, \mathrm{waiting}), r, \cdot, \cdot, \cdot, \cdot\big)$$

2. *If $\mathbf{C}_u.\mathsf{P1}^r_{i\to\mathrm{lead}(r)} \neq \bot$, for some $u \in T$, $i \in \mathbb{P}$, and $r > 0$, then there is $t \in T$, $t \leq u$, for which*

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{i:(\text{RCV-PREP-REQ})} \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle$$

   such that, for some $b \in \mathbb{V} \times \mathbb{N}$ and $p < r$,

$$\begin{array}{rcl@{\qquad}rcl} \mathbf{C}_{t-1}.\mathsf{P1}^r_{i\to\mathrm{lead}(r)} & = & \bot & \mathbf{S}_{t-1}(i) & = & \big(\cdot, \cdot, p, b, \cdot, \cdot\big) \\ \mathbf{C}_t.\mathsf{P1}^r_{i\to\mathrm{lead}(r)} & = & (b, (\iota, 0, 0, 0)) & \mathbf{S}_t(i) & = & \big(\cdot, \cdot, r, b, \cdot, \cdot\big) \end{array}$$

3. *If $\mathbf{C}_u.\mathsf{P2}^r_{\mathrm{lead}(r)\to k} \neq \bot$, for some $u \in T$, $k \in \mathbb{P}$, and $r > 0$, then there is $t \in T$, $t \leq u$, for which*

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{\mathrm{lead}(r):(\text{SND-ACC-REQ})} \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle$$

such that for all $i \in \mathbb{P}$

$$\mathbf{C}_{t-1}.\mathsf{P2}^{r}_{\text{lead}(r)\to i} = \bot \qquad \mathbf{S}_{t-1}(\text{lead}(r)) = \big((\text{yes}, \text{waiting}), \cdot, \cdot, \cdot, \cdot, \cdot\big)$$

$$\mathbf{C}_{t}.\mathsf{P2}^{r}_{\text{lead}(r)\to i} = (\text{best}^{r}_{\iota}(\mathbf{C}_{t-1}), (\iota, 0, 0, 0))$$

$$\mathbf{S}_{t}(\text{lead}(r)) = \big((\text{yes}, \text{polling}), \cdot, \cdot, \cdot, \cdot, \cdot\big)$$

4. *If* $\mathbf{C}_u.\mathsf{P3}^{p}_{i\to\text{lead}(p)} \neq \bot$*, for some* $u \in T$*,* $i \in \mathbb{P}$*, and* $p > 0$*, then there is* $t \in T$*,* $t \leq u$*, for which*

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \ \to_{i:(\text{RCV-ACC-REQ})} \langle\, \mathbf{B}_{t}, \mathbf{C}_{t}, \mathbf{S}_{t} \,\rangle$$

such that, for some $(v_i, s_i) \in \mathbb{V} \times \mathbb{N}$*,* $p \in \mathbb{N}$ *with* $s_i < p$*:*

$$\mathbf{C}_{t-1}.\mathsf{P3}^{p}_{i\to\text{lead}(p)} = \bot \qquad \mathbf{S}_{t-1}(i) = \big(\cdot, \cdot, p, (v_i, s_i), \cdot, \cdot\big)$$

$$\mathbf{C}_{t}.\mathsf{P3}^{p}_{i\to\text{lead}(p)} = (\text{val}^{p}(\mathbf{C}_{t-1}), (\iota, 0, 0, 0))$$

$$\mathbf{S}_{t}(i) = \big(\cdot, \cdot, p, (\text{val}^{p}(\mathbf{C}_{t-1}), p), \cdot, \cdot\big)$$

where $\text{val}^{p}(\mathbf{C}_{t-1}) \neq \bot$*.*

5. *If* $\mathbf{B}_u(r) \neq \bot$*, for some* $u \in T$ *and for some* $r > 0$*, then there is* $t \in T$*,* $t \leq u$*, for which*

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \ \to_{\text{lead}(r):(\text{SUCCESS})} \langle\, \mathbf{B}_{t}, \mathbf{C}_{t}, \mathbf{S}_{t} \,\rangle$$

such that, for some $v \in \mathbb{V}$

$$\begin{array}{rcl} \mathbf{B}_{t-1}(r) & = & \bot \\ \mathbf{B}_{t}(r) & = & (v, \mathbb{P}) \end{array} \qquad \begin{array}{rcl} \mathbf{S}_{t-1}(\text{lead}(r)) & = & \big((\text{yes}, \text{polling}), r, \cdot, \cdot, \cdot, \cdot\big) \\ \mathbf{S}_{t}(\text{lead}(r)) & = & \big((\text{yes}, \text{done}), r, \cdot, \cdot, \cdot, \cdot\big)\,. \end{array}$$

Proof. As to the first four statements, since $\mathbf{C}_0 = \emptyset$, and since $\mathbf{C}.X \neq \bot$ (with X equals to $\mathsf{P0}^{r}_{\text{lead}(r)\to i}$ for (1), $\mathsf{P1}^{r}_{i\to\text{lead}(r)}$ for (2), $\mathsf{P2}^{r}_{\text{lead}(r)\to i}$ for (3), $\mathsf{P3}^{r}_{i\to\text{lead}(r)}$ for (4)) we can infer that entry X must have been added during the run. The only way of adding CH-messages to the history during a run is by executing a specific transition rule ((SND-PREP-REQ) for (1), (RCV-PREP-REQ) for (2), (SND-ACC-REQ) for (3) or (RCV-ACC-REQ) for (4)). Note that, even if we suppose that the very same entry can be added more than once per run (message reproduction), since $\mathbf{C}_0 = \emptyset$ we can always identify and choose a point in the run where the entry is introduced for the first time. The resulting data follows from the analysis of the premises and the conclusions of each of those rules.

Few more words need to be spent for the conclusion $\mathrm{val}^p(\mathbf{C}_{t-1}) \neq \bot$ in 4, since it does not appear explicitly in the rules. From its definition, we know that $\mathrm{val}^p(\mathbf{C}_{t-1}) = v$ if for all $j \in \mathbb{P}$ it holds that $\mathbf{C}_{t-1}.\mathsf{P2}^p_{\mathrm{lead}(p)\to j} = ((v, \cdot), \cdot)$.
From the premises of rule (RCV-ACC-REQ), we know that $\mathbf{C}_{t-1}.\mathsf{P2}^p_{\mathrm{lead}(p)\to i} = ((v, s), (\cdot, \cdot, \cdot, \delta))$ with $\delta \geq 1$ for some $i \in \mathbb{P}$. However, all the 2nd-phase messages of a request number p are created at the same time, by $\mathrm{lead}(p)$, by performing the rule (SND-ACC-REQ). As a consequence, for all $j \in \mathbb{P}$, it holds that $\mathbf{C}_{t-1}.\mathsf{P2}^p_{\mathrm{lead}(p)\to j} = ((v, \cdot), \cdot)$ (even if some j crashed). Thus, $\mathrm{val}^p(\mathbf{C}_{t-1})$ is defined.

The proof in the broadcast case (5) is completely analogous to the previous ones with respect to the choice of the first addition (out of possibly several) of a broadcast message, but instead using the rule (SUCCESS) to directly read off the relevant data for $\mathbf{B}_t(r)$, $\mathbf{S}_{t-1}(\mathrm{lead}(r))$ and $\mathbf{S}_t(\mathrm{lead}(r))$.
□

An important property of the Consensus algorithm is that both CH- and RBC-messages are unique. This is guaranteed by the standard technique of adding enough distinguishing information to messages. Intuitively, we formalize the absence of message reproduction by stating that *every addition of a message is fresh*. More precisely, whenever in a run a computation step is derived using one of the five rules that *add* a message to some component $\mathbf{C}$ or $\mathbf{B}$, then the respective entry *must* have been undefined up to this very moment. An alternative and slightly more technical interpretation of this result is: if we had included conditions to ensure the prior non-definedness of added messages as premises to the respective rules, then those conditions would have been redundant.

Lemma 9.1.2.3 (Absence of Message Reproduction) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run such that, for some* $z \in T$,

$$\langle\, \mathbf{B}_{z-1}, \mathbf{C}_{z-1}, \mathbf{S}_{z-1} \,\rangle \rightarrow_{i:(\textsc{Rule})} \langle\, \mathbf{B}_z, \mathbf{C}_z, \mathbf{S}_z \,\rangle$$

with $\mathbf{S}_{z-1}(i) = (\cdot, r, p, \cdot, \cdot, \cdot)$, *for some* $r, p \in \mathbb{N}$, $r, p > 0$. *Then,*

1. *If* (RULE) *is* (SND-PREP-REQ), *then* $\mathbf{C}_{z-1}.\mathsf{P0}^r_{\mathrm{lead}(r)\to k} = \bot$.
2. *If* (RULE) *is* (RCV-PREP-REQ), *then* $\mathbf{C}_{z-1}.\mathsf{P1}^r_{i\to\mathrm{lead}(r)} = \bot$.
3. *If* (RULE) *is* (SND-ACC-REQ), *then* $\mathbf{C}_{z-1}.\mathsf{P2}^r_{\mathrm{lead}(r)\to k} = \bot$.
4. *If* (RULE) *is* (RCV-ACC-REQ), *then* $\mathbf{C}_{z-1}.\mathsf{P3}^p_{i\to\mathrm{lead}(p)} = \bot$.
5. *If* (RULE) *is* (SUCCESS), *then* $\mathbf{B}_{z-1}(r) = \bot$.

Proof. The main reasoning is done by contradiction.

1. If (RULE) is (SND-PREP-REQ), then $\mathbf{S}_{z-1}(i) = \big((\text{yes}, \text{trying}), r, \cdot, \cdot, (\top, \iota), \cdot\big)$. Suppose by contradiction that $\mathbf{C}_{z-1}.\mathsf{P0}^{r}_{\text{lead}(r)\to i} \neq \bot$. By an application of Lemma 9.1.2.2(1) there is $t \in T$, $t \leq z{-}1$, such that

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{i:(\text{SND-PREP-REQ})} \langle\, \mathbf{B}_{t}, \mathbf{C}_{t}, \mathbf{S}_{t} \,\rangle$$

with $\mathbf{S}_{t-1}(i) = \big((\text{yes}, \text{trying}), r, \cdot, \cdot, (\top, \iota), \cdot\big)$ and $\mathbf{S}_{t}(i) = \big((\text{yes}, \text{waiting}), r, \cdot, \cdot, (\top, \iota), \cdot\big)$. By Lemma 9.1.2.1(5), the execution of rule (SND-PREP-REQ) implies $\mathbf{S}_{t-1}(i) < \mathbf{S}_{t}(i)$. On the other hand, since $t \leq z-1$ and since process states are non-decreasing (Lemma 9.1.2.1), we have $\mathbf{S}_{t}(i) \leq \mathbf{S}_{z-1}(i)$. We have reached a contradiction. Thus, the assumption $\mathbf{C}_{z-1}.\mathsf{P0}^{r}_{\text{lead}(r)\to i} \neq \bot$ must have been wrong;
2. If (RULE) is (RCV-PREP-REQ), then $\mathbf{S}_{z-1}(i) = \big(\cdot, \cdot, p, \cdot, (\top, \iota), \cdot\big)$. Suppose by contradiction that $\mathbf{C}_{z-1}.\mathsf{P1}^{r}_{i\to\text{lead}(r)} \neq \bot$. By an application of Lemma 9.1.2.2(2) there is $t \in T$, $t \leq z{-}1$, such that

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{i:(\text{RCV-PREP-REQ})} \langle\, \mathbf{B}_{t}, \mathbf{C}_{t}, \mathbf{S}_{t} \,\rangle$$

with $p_{t-1} < r$, $\mathbf{S}_{t-1}(i) = \big(\cdot, \cdot, p_{t-1}, \cdot, (\top, \iota), \cdot\big)$ and $\mathbf{S}_{t}(i) = \big(\cdot, \cdot, r, \cdot, (\top, \iota), \cdot\big)$. By Lemma 9.1.2.1(6), the execution of rule (RCV-PREP-REQ) implies $\mathbf{S}_{t-1}(i) < \mathbf{S}_{t}(i)$. On the other hand, since $t \leq z-1$ and since process states are non-decreasing (Lemma 9.1.2.1), we have $\mathbf{S}_{t}(i) \leq \mathbf{S}_{z-1}(i)$. We have reached a contradiction.
Thus, the assumption $\mathbf{C}_{z-1}.\mathsf{P1}^{r}_{i\to\text{lead}(r)} \neq \bot$ must have been wrong;
3. Completely analogous to the first case, but using Lemma 9.1.2.2(3);
4. Analogous to the second case, but using Lemma 9.1.2.2(4);
5. Analogous to the previous cases, but instead using Lemma 9.1.2.2(5).

□

Notice that none of these proofs refers to a particular incarnation of the sending process: the messages are UNIQUE in the run, in the sense that not even a different incarnation of the same process can reproduce the same message.

Putting it all together, we can now state the monotonicity results for the two kinds of messages.

Lemma 9.1.2.4 (Monotonicity of Message Histories) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)_{T}^{(v_1,\ldots,v_n)}$ *be a run such that*

$$\langle\, \mathbf{B}_{u-1}, \mathbf{C}_{u-1}, \mathbf{S}_{u-1} \,\rangle \;\rightarrow_{i:(\text{RULE})} \langle\, \mathbf{B}_{u}, \mathbf{C}_{u}, \mathbf{S}_{u} \,\rangle$$

for some $u \in T$ *and* $i \in \mathbb{P}$.

1. *If* (RULE) *is any of the rules* (NEW), (SUCCESS), (RBC-DELIVER), (LEADER-YES), (LEADER-NO), (CRASH) *or* (RECOVERY), *then* $\mathbf{C}_{u-1} = \mathbf{C}_u$.
2. *If* (RULE) *is any of the rules* (SND-PREP-REQ), (RCV-PREP-REQ), (SND-ACC-REQ), (RCV-ACC-REQ), (CH_SND), (CH_DELIVER), (CH_DUPL) *or* (CH_LOSS), *then* $\mathbf{C}_{u-1} < \mathbf{C}_u$.
3. *If* (RULE) *is any of the rules* (SUCCESS) *or* (RBC-DELIVER), *then* $\mathbf{B}_{u-1} < \mathbf{B}_u$.
4. *If* (RULE) *is any of the rules* (NEW), (SND-PREP-REQ), (RCV-PREP-REQ), (SND-ACC-REQ), (RCV-ACC-REQ), (CH_SND), (CH_DUPL) (CH_LOSS), (CH_DELIVER), (LEADER-YES), (LEADER-NO), (CRASH) *or* (RECOVERY), *then* $\mathbf{B}_{u-1} = \mathbf{B}_u$.

Proof. We prove the result by local comparison of premises and conclusions of each possible rule (RULE).
If (RULE) is any of the rules (NEW), (SUCCESS), (RBC-DELIVER), (LEADER-YES), (LEADER-NO), (CRASH) or (RECOVERY) the $\mathbf{C}$ component is not touched, and $\mathbf{C}_{u-1} = \mathbf{C}_u$.
Similarly, if (RULE) is any of the rules (NEW), (SND-PREP-REQ), (RCV-PREP-REQ), (SND-ACC-REQ), (RCV-ACC-REQ), (CH_SND), (CH_DELIVER), (CH_DUPL), (CH_LOSS), (LEADER-YES), (LEADER-NO), (CRASH) or (RECOVERY), the $\mathbf{B}$ component is not touched, and $\mathbf{B}_{u-1} = \mathbf{B}_u$.
If (RULE) is (SND-PREP-REQ), (RCV-PREP-REQ), (SND-ACC-REQ), (RCV-ACC-REQ), then we know (Lemma 9.1.2.3) that $\mathbf{C}_{u-1}.X = \bot$, while the application of (RULE) produces $\mathbf{C}_u.X \neq \bot$ with $X \in \{ \mathsf{P0}^r_{j\to k}, \mathsf{P1}^r_{j\to k}, \mathsf{P2}^r_{j\to k}, \mathsf{P3}^r_{j\to k} \}$ respectively. Since $\mathbf{C}_{u-1} \neq \mathbf{C}_u$, by Definition 9.1.1.4 we can say that $\mathbf{C}_{u-1} < \mathbf{C}_u$.
If (RULE) is (CH_SND), (CH_DELIVER), (CH_DUPL) or (CH_LOSS), then the message tag is increased and by Definition 9.1.1.4 we can conclude $\mathbf{C}_{u-1} < \mathbf{C}_u$.
If (RULE) is (SUCCESS) or (RBC-DELIVER), then a new (Lemma 9.1.2.3) message is broadcasted or an already existing message is delivered. By Definition 9.1.1.5 we have that $\mathbf{B}_{u-1} \leq \mathbf{B}_u$, and, since $\mathbf{B}_{u-1} \neq \mathbf{B}_u$, we conclude $\mathbf{B}_{u-1} < \mathbf{B}_u$. □

Finally, we give here two straightforward results on the monotonicity of message histories. The first states that the *contents* of messages that are stored within $\mathbf{C}$ never change; only the transmission *tag* may change. The second states that round proposals never get overwritten by later updates of $\mathbf{C}$.

Corollary 9.1.2.5 *Let* $R = \big(\langle \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run and* $X \in \{ \mathsf{P0}^r_{j\to k}, \mathsf{P1}^r_{j\to k}, \mathsf{P2}^r_{j\to k}, \mathsf{P3}^r_{j\to k} \}$ *, where* $r \in \mathbb{N}$*,* $j, k \in \mathbb{P}$*. Let* $t, z \in T$ *with* $t \leq z$*.*

1. *If $\mathbf{C}_t.X = (V, \tau)$, then there is τ' such that $\mathbf{C}_z.X = (V, \tau')$ with $\tau \leq \tau'$.*
2. *If $\mathbf{B}_t = (v, D)$, then there is D' such that $\mathbf{B}_z = (v, D')$ with $D \supseteq D'$.*

Proof.

1. By Definition 9.1.1.4 and iterated application of Lemma 9.1.2.4.
2. By Definition 9.1.1.5 and iterated application of Lemma 9.1.2.4.

□

Corollary 9.1.2.6 *Let $R = \big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ be a run. Let $\mathrm{val}^r(\mathbf{C}_t) \neq \bot$, for some $t \in T$ and round $r > 0$, then for all $z \in T$, $t \leq z$, it holds that $\mathrm{val}^r(\mathbf{C}_z) = \mathrm{val}^r(\mathbf{C}_t)$.*

Proof. By the definition of $\mathrm{val}^r(\mathbf{C})$ and Corollary 9.1.2.5 for the case 2nd-phase messages. □

In the following lemma we prove that the stamp contained in the current belief of a process state is always smaller than or equal to the current prepare request number.

Lemma 9.1.2.7 (Stamp consistency in process states) *Let $\big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ be a run.*
Let $t \in T$ and $i \in \mathbb{P}$ with $\mathbf{S}_t(i) = \big(\cdot, r_t, p_t, (\cdot, s_t), (u_t, \iota_t), \cdot\big)$.

1. *then $s_t \leq p_t$.*
2. *If $z \in T$, with $t \leq z$ and $\mathbf{S}_z(i) = \big(\cdot, r_z, p_z, (\cdot, s_z), (u_z, \iota_z), \cdot\big)$, then $s_t \leq s_z$.*

Proof.

1. We do it by induction on $t \in T$, corresponding to the length of the prefix $\langle\, \mathbf{B}_0, \mathbf{C}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \,\rangle \rightarrow^t \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle$ of the given run.

   Base case $t = 0$ **:** By definition, for all $i \in \mathbb{P}$, we have
   $\mathbf{S}_0(i) = \big((\mathrm{no}, \mathrm{idle}), 0, 0, (v_i, 0), (\top, 1), \bot\big)$.

   Inductive case $t > 0$ **:** We analyze the step

   $$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \rightarrow_{j:(\textsc{Rule})} \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle$$

   of the given run, with $\mathbf{S}_{t-1}(i) = \big(\cdot, r_{t-1}, p_{t-1}, (\cdot, s_{t-1}), (u_{t-1}, \iota_{t-1}), \cdot\big)$ and $\mathbf{S}_t(i) = \big(\cdot, r_t, p_t, (\cdot, s_t), (u_t, \iota_t), \cdot\big)$
   If $i \neq j$, then the desired stamp consistency statement is immediately true by the induction hypothesis, because the last transition only touches the state of process j. If $i = j$, then we proceed by case analysis on rule (Rule), keeping in mind that, by inductive hypothesis, we have $s_{t-1} \leq p_{t-1}$. (Rule) can be

(a) a rule that does not affect the prepare request number or the estimate of process i (i.e.,(CRASH), (RECOVERY), (LEADER-YES), (LEADER-NO), (CH_SND), (CH_DELIVER), (CH_DUPL), (CH_LOSS), (NEW), (SND-PREP-REQ), (SND-ACC-REQ), (SUCCESS), (RBC-DELIVER)), so $s_t = s_{t-1} \leq p_{t-1} = p_t$;

(b) rule (RCV-PREP-REQ), so there is a $r \in \mathbb{N}$ with $p_{t-1} < r$ such that $p_t = r$; since $b_{t-1} = b_t$, we have $s_t = s_{t-1} \leq p_{t-1} \leq r = p_t$;

(c) rule (RCV-ACC-REQ) where $p_{t-1} = p_t$, s_t is set equal to p_{t-1} and therefore $s_t \leq p_{t-1} \leq p_t$.

2. If $t = z$ then $s_t = s_z$. If $t < z$ then we will prove that for all $w \in T$, with $t{\leq}w{<}z$, it holds that $s_w{\leq}s_{w+1}$. Thanks to the transitivity properties of $<$, we can prove our claim through an iteration of this result for all w. Let us fix $w \in T$ such that $t{\leq}w{<}z$. Now, consider the transition

$$\langle\, \mathbf{B}_w, \mathbf{C}_w, \mathbf{S}_w \,\rangle \;\rightarrow_{j:(\text{RULE})} \langle\, \mathbf{B}_{w+1}, \mathbf{C}_{w+1}, \mathbf{S}_{w+1} \,\rangle$$

for $j \in \mathbb{P}$. If $j \neq i$ then $s_w{=}s_{w+1}$. Suppose now $j{=}i$. By inspection on the transition rules of Tables 7.1 and 7.2:

- If (RULE) is any rule but (RCV-ACC-REQ) then the belief component of the state remains unchanged, and hence $s_w{=}s_{w+1}$.
- If (RULE)=(RCV-ACC-REQ) then process i adopts the proposal with the current prepare request number as stamp, and hence $s_{w+1}{=}p_w$. By the premises of the rule, we know that $s_w{<}p_w$, hence $s_w{\leq}p_w{=}s_{w+1}$.

□

The previous consistency lemma for process state information directly carries over to messages.

Lemma 9.1.2.8 (Stamp consistency in process proposal) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run.*
Let $\mathbf{C}_z.\mathsf{P1}^r_{i\to\text{lead}(r)}{=}((\cdot, s), \cdot)$ *for some* $z \in T$*, proposal number* $r \in \mathbb{N}$*, stamp* $s \in \mathbb{N}$ *and process* $i \in \mathbb{P}$*. Then:*

1. $s < r$
2. *for all* r' *with* $0 < r' < r$ *for which* $\mathbf{C}_z.\mathsf{P1}^{r'}_{i\to\text{lead}(r')} \neq \bot$*, we have* $\mathbf{C}_z.\mathsf{P1}^{r'}_{i\to\text{lead}(r')}{=}((\cdot, s'), \cdot)$*, for some* $s' \in \mathbb{N}$ *with* $s' \leq s$.

Proof.

1. 1st-phase messages can only be created by applying rule (RCV-PREP-REQ), where the belief component of the current state is copied into the new 1st-phase message. As rule (RCV-PREP-REQ) can only be applied if the process' last prepare request number is $p \in \mathbb{N}$ with $p < r$, by Lemma 9.1.2.7(1) it follows that $s \leq p < r$.
2. As $\mathbf{C}_z.\mathsf{P1}^{r}_{i\to\mathrm{lead}(r)}{=}((\cdot, s), \cdot)$, by Lemma 9.1.2.2(2) there is $u \in T$ with $u \leq z$, for which

   $$\langle\, \mathbf{B}_{u-1}, \mathbf{C}_{u-1}, \mathbf{S}_{u-1} \,\rangle \;\rightarrow_{i:(\text{RCV-PREP-REQ})}\; \langle\, \mathbf{B}_{u}, \mathbf{C}_{u}, \mathbf{S}_{u} \,\rangle$$

   where $\mathbf{S}_{u-1}(i){=}\big(\cdot, r_i, p, (\cdot, s), (\top, \iota), \cdot\big)$ and $\mathbf{S}_{u}(i){=}\big(\cdot, r_i, r, (\cdot, s), (\top, \iota), \cdot\big)$. Let us fix an arbitrary $r' < r$ such that $\mathbf{C}_u.\mathsf{P1}^{r'}_{i\to\mathrm{lead}(r')} \neq \bot$. Let $\mathbf{C}_u.\mathsf{P1}^{r'}_{i\to\mathrm{lead}(r')}{=}((\cdot, s'), \cdot)$ for some $s' \in \mathbb{N}$. Again, by Lemma 9.1.2.2(2) there is $t \in T$ for which

   $$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{i:(\text{RCV-PREP-REQ})}\; \langle\, \mathbf{B}_{t}, \mathbf{C}_{t}, \mathbf{S}_{t} \,\rangle$$

   where $\mathbf{S}_{t-1}(i){=}\big(\cdot, r'_i, p', (\cdot, s'), (\top, \iota'), \cdot\big)$ and $\mathbf{S}_{t}(i){=}\big(\cdot, r'_i, r', (\cdot, s'), (\top, \iota'), \cdot\big)$, with $p' < r'$. Since $r' < r$, by Definition 9.1.1.1 we have $(\cdot, r'_i, r') <_i (\cdot, r_i, r)$; by Definition 9.1.1.2 it follows that $\mathbf{S}_t(i) < \mathbf{S}_u(i)$.
   By Lemma 9.1.2.1 process states are non-decreasing, and hence $t < u$. By Lemma 9.1.2.7(2), we derive $s' \leq s$. By Corollary 9.1.2.5 and $\mathbf{C}_u.\mathsf{P1}^{r'}_{i\to\mathrm{lead}(r')}{=}((\cdot, s'), \cdot)$, we also have $\mathbf{C}_z.\mathsf{P1}^{r'}_{i\to\mathrm{lead}(r')}{=}((\cdot, s'), \cdot)$.

□

A process is not obliged to participate to all the rounds of the algorithm. We define here a function that, given a run R and a proposal number r, returns the first proposal number greater than r to which process i answers in run R.

Definition 9.1.2.9 *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)^{(v_1,\ldots,v_n)}_{T}$ *denote a run. Let* min *denote the standard operator to calculate the minimum of a set of integers. Let*

$$\text{acc-prep}_i(R) \;\overset{\text{def}}{=}\; \{\; r \in \mathbb{N} \mid \exists u \in T \;\textit{ such that }\; \mathbf{C}_u.\mathsf{P1}^{r}_{i\to\mathrm{lead}(r)} \neq \bot \;\}$$

denote the set of all requests numbers in run R *in which process* i *has answered by sending its estimate. Let*

$$\mathrm{f}^{r}_{i}(R) \;\overset{\text{def}}{=}\; \min\{\; \hat{r} \in \text{acc-prep}_i(R) \mid \hat{r} > r \;\}$$

denote the first proposal number greater than r *to which process* i *answers in run* R*.*

The next lemma states how the proposal of a process, i.e., its $\mathsf{P1}$ message, correlates with the $\mathsf{P3}$ message that the process itself can have sent during the last prepare request number in which it has participated. In fact, the $\mathsf{P3}$ message is an acknowledgment for the proposal: depending on whether the process sends the message or not, it adopts the proposal as its own new belief or it keeps the old one. It is this possibly updated belief that the process sends as its proposal when answering to the following prepare request.

Lemma 9.1.2.10 (Proposal creation) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run. Let* $\mathbf{C}_z.\mathsf{P1}^r_{i\to\mathrm{lead}(r)} = ((v,s),\cdot)$ *and* $\mathbf{C}_z.\mathsf{P1}^{\mathrm{f}_i^r(R)}_{i\to\mathrm{lead}(\mathrm{f}_i^r(R))} = ((v',s'),\cdot)$*, for some* $z \in T$*, round* $r > 0$*, values* $v, v' \in \mathbb{V}$*, and stamps* $s, s' \in \mathbb{N}$*.*

1. *If* $\mathbf{C}_z.\mathsf{P3}^r_{i\to\mathrm{lead}(r)} = \bot$*, then* $v = v'$ *and* $s = s'$*.*
2. *If* $\mathbf{C}_z.\mathsf{P3}^r_{i\to\mathrm{lead}(r)} \neq \bot$*, then* $v' = \mathrm{val}^r(\mathbf{C}_z)$ *and* $s < s' = r$*.*

Proof.

1. By inspection of the rules of Tables 7.1 and 7.2, the only rule that can change the estimate is (RCV-ACC-REQ). Let us call $u \in T$ the time at which

$$\langle\, \mathbf{B}_{u-1}, \mathbf{C}_{u-1}, \mathbf{S}_{u-1} \,\rangle \ \to_{i:(\text{RCV-PREP-REQ})} \langle\, \mathbf{B}_u, \mathbf{C}_u, \mathbf{S}_u \,\rangle$$

such that

$$\begin{array}{rclrcl} \mathbf{C}_{u-1}.\mathsf{P1}^r_{i\to\mathrm{lead}(r)} & = & \bot & \mathbf{S}_{u-1}(i) & = & \big(\cdot,\cdot,p,(v,s),(\top,\cdot),\cdot\big) \\ \mathbf{C}_u.\mathsf{P1}^r_{i\to\mathrm{lead}(r)} & = & ((v,s),\cdot) & \mathbf{S}_u(i) & = & \big(\cdot,\cdot,r,(v,s),(\top,\cdot),\cdot\big) \end{array}$$

and let us call $u' \in T$ the time at which

$$\langle\, \mathbf{B}_{u'-1}, \mathbf{C}_{u'-1}, \mathbf{S}_{u'-1} \,\rangle \ \to_{i:(\text{RCV-PREP-REQ})} \langle\, \mathbf{B}_{u'}, \mathbf{C}_{u'}, \mathbf{S}_{u'} \,\rangle$$

such that

$$\mathbf{C}_{u'-1}.\mathsf{P1}^{\mathrm{f}_i^r(R)}_{i\to\mathrm{lead}(\mathrm{f}_i^r(R))} = \bot \qquad \mathbf{S}_{u'-1}(i) = \big(\cdot,\cdot,r,(v',s'),(\top,\cdot),\cdot\big)$$

$$\mathbf{C}_{u'}.\mathsf{P1}^{\mathrm{f}_i^r(R)}_{i\to\mathrm{lead}(\mathrm{f}_i^r(R))} = ((v',s'),\cdot)$$

$$\mathbf{S}_{u'}(i) = \big(\cdot,\cdot,\mathrm{f}_i^r(R),(v',s'),(\top,\cdot),\cdot\big)$$

By Lemma 9.1.2.1 process states are non-decreasing, we have $u \leq u' \leq z$. Since $\mathbf{C}_z.\mathsf{P3}^r_{i\to\mathrm{lead}(r)} = \bot$, we know that for all $t \in T$, with $u \leq t \leq u' \leq z$, such that

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \ \to_{j:(\text{RULE})} \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle$$

either $j \neq i$ or $(\text{RULE}) \neq (\text{RCV-ACC-REQ})$. Given that (RCV-ACC-REQ) is the only rule that can change the estimate of a process, and given that such rule has not been applied between time u and time u', we conclude that it must be $v' = v$ and $s' = s$.

2. As $\mathbf{C}_z.\mathsf{P3}^r_{i\to\text{lead}(r)} \neq \bot$, by Lemma 9.1.2.2(4) there is $t \in T$, $t \leq z$, for which

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{i:(\text{RCV-ACC-REQ})} \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle$$

such that

$$\mathbf{C}_{t-1}.\mathsf{P3}^r_{i\to\text{lead}(r)} = \bot \qquad\qquad \mathbf{S}_{t-1}(i) = \big(\cdot,\cdot,r,(v,s_i),(\top,\cdot),\cdot\big)$$

$$\mathbf{C}_t.\mathsf{P3}^r_{i\to\text{lead}(r)} = (\text{val}^r(\mathbf{C}_{t-1}),\cdot)$$

$$\mathbf{S}_t(i) = \big(\cdot,\cdot,r,(\text{val}^r(\mathbf{C}_{t-1}),r),(\top,\cdot),\cdot\big)$$

As $\mathbf{C}_z.\mathsf{P1}^{\text{f}^r_i(R)}_{i\to\text{lead}(\text{f}^r_i(R))} = \big((v',s'),\cdot\big)$, by Lemma 9.1.2.2(2) there is $u \in T$, $u \leq z$, for which

$$\langle\, \mathbf{B}_{u-1}, \mathbf{C}_{u-1}, \mathbf{S}_{u-1} \,\rangle \;\rightarrow_{i:(\text{RCV-PREP-REQ})} \langle\, \mathbf{B}_u, \mathbf{C}_u, \mathbf{S}_u \,\rangle$$

such that,

$$\mathbf{C}_{u-1}.\mathsf{P1}^{\text{f}^r_i(R)}_{i\to\text{lead}(\text{f}^r_i(R))} = \bot \qquad\qquad \mathbf{S}_{u-1}(i) = \big(\cdot,\cdot,r,(v',s'),(\top,\cdot),\cdot\big)$$

$$\mathbf{C}_u.\mathsf{P1}^{\text{f}^r_i(R)}_{i\to\text{lead}(\text{f}^r_i(R))} = \big((v',s'),\cdot\big)$$

$$\mathbf{S}_u(i) = \big(\cdot,\cdot,\text{f}^r_i(R),(v',s'),(\top,\cdot),\cdot\big)$$

By Lemma 9.1.2.1 process states are non-decreasing. As $\mathbf{S}_t(i) < \mathbf{S}_u(i)$, it follows that $t < u$, and the two transitions are ordered in time as follows:

$$\begin{array}{lll}
\langle\, \mathbf{B}_0, \mathbf{C}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \,\rangle & \rightarrow^* & \langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \\
 & \rightarrow_{i:(\text{RCV-ACC-REQ})} & \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle \\
 & \rightarrow^* & \langle\, \mathbf{B}_{u-1}, \mathbf{C}_{u-1}, \mathbf{S}_{u-1} \,\rangle \\
 & \rightarrow_{i:(\text{RCV-PREP-REQ})} & \langle\, \mathbf{B}_u, \mathbf{C}_u, \mathbf{S}_u \,\rangle \\
 & \rightarrow^* & \langle\, \mathbf{B}_z, \mathbf{C}_z, \mathbf{S}_z \,\rangle
\end{array}$$

By Lemma 9.1.2.1 process states are non-decreasing. As a consequence, the only rules involving process i that can have been applied in the derivation sequence $\langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle \;\rightarrow^* \langle\, \mathbf{B}_{u-1}, \mathbf{C}_{u-1}, \mathbf{S}_{u-1} \,\rangle$ and that can possibly have affected its state are: (NEW), (SND-PREP-REQ), (SND-ACC-REQ), (SUCCESS), (RBC-DELIVER), (CRASH), (RECOVERY), (LEADER-YES) and (LEADER-NO). However, none of these rules affects the belief component of the state, and hence $(v', s'){=}(\mathrm{val}^r(\mathbf{C}_{t-1}), r)$. (Note that (RCV-ACC-REQ) could not have been applied because it requires the stamp of the process belief to be strictly smaller than the prepare request number, while in $\mathbf{S}_t(i)$ these two componenets are both equal to r and both $\mathbf{S}_t(i)$ and $\mathbf{S}_{u-1}(i)$ have request number equal to r.) By Corollary 9.1.2.6 $\mathrm{val}^r(\mathbf{C}_{t-1}){=}\,\mathrm{val}^r(\mathbf{C}_z)$. As a consequence, $v' = \mathrm{val}^r(\mathbf{C}_z)$ and $s' = r$. Finally, since $\mathbf{C}_z.\mathsf{P1}^r_{i\to\mathrm{lead}(r)} = ((v, s), \cdot)$, by Lemma 9.1.2.8(1) it follows that $s < r$.

□

We need a last technical lemma in order to show how process proposals are generated.

Lemma 9.1.2.11 *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run. Let* $z \in T$, $r, r' \in \mathbb{N}$ *with* $0 < r' < r$, *and* $i \in \mathbb{P}$ *such that* $\mathbf{C}_z.\mathsf{P1}^{r'}_{i\to\mathrm{lead}(r')}{=}((v', s'), \cdot)$ *and* $\mathbf{C}_z.\mathsf{P1}^{r}_{i\to\mathrm{lead}(r)}{=}((v, s), \cdot)$.

1. *If* $s' = s$ *then* $v' = v$ *and for all* $\hat{r}$, $r' \le \hat{r} < r$, *it holds that* $\mathbf{C}_z.\mathsf{P3}^{\hat{r}}_{i\to\mathrm{lead}(\hat{r})}{=}\bot$.
2. *If* $s' < s$ *then there is a proposal number* $\hat{r}$, $r'{\le}\hat{r}{<}r$, *such that* $\mathbf{C}_z.\mathsf{P3}^{\hat{r}}_{i\to\mathrm{lead}(\hat{r})}{=}(v, \cdot)$.

Proof. As $\mathbf{C}_z.\mathsf{P1}^{r}_{i\to\mathrm{lead}(r)}{=}((v, s), \cdot)$, by Lemma 9.1.2.2(2) there is $u \in T$, $u \le z$, for which

$$\langle\, \mathbf{B}_{u-1}, \mathbf{C}_{u-1}, \mathbf{S}_{u-1} \,\rangle \ \to_{i:(\text{RCV-PREP-REQ})} \langle\, \mathbf{B}_u, \mathbf{C}_u, \mathbf{S}_u \,\rangle$$

with $\mathbf{S}_{u-1}(i) = \big(\cdot, \cdot, p, (v, s), \cdot, \cdot\big)$ and $\mathbf{S}_u(i) = \big(\cdot, \cdot, r, (v, s), \cdot, \cdot\big)$.

As $\mathbf{C}_z.\mathsf{P1}^{r'}_{i\to\mathrm{lead}(r')}{=}((v', s'), \cdot)$, by Lemma 9.1.2.2(2) there is $t \in T$, $t \le z$, for which

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \ \to_{i:(\text{RCV-PREP-REQ})} \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle$$

with $\mathbf{S}_{t-1}(i) = \big(\cdot, \cdot, p', (v', s'), \cdot, \cdot\big)$ and $\mathbf{S}_t(i) = \big(\cdot, \cdot, r', (v', s'), \cdot, \cdot\big)$. Since $r' < r$, by Lemma 9.1.2.1 we have $t < u$. If $t = u{-}1$, then $s' = s$, $v' = v$ and obviously there is no $\hat{r}$, $r' \le \hat{r} < r$ for which $\mathbf{C}_z.\mathsf{P3}^{\hat{r}}_{i\to\mathrm{lead}(\hat{r})} \neq \bot$. Let us now assume $t < u{-}1$ and examine what can have happened between t and $u - 1$.

1. If $s' = s$, then by Lemma 9.1.2.8(1) we know that $s' = s < r' < r$. Let us suppose by contradiction that there is a $\hat{r}$, with $r'{\le}\hat{r}{<}r$, for which

$\mathbf{C}_z.\mathsf{P3}^{\hat{r}}_{i\to\text{lead}(\hat{r})} = (\hat{v}, \cdot)$. In this case, by Lemma 9.1.2.2(4), there is a $\hat{u} \in T$ such that

$$\langle\, \mathbf{B}_{\hat{u}-1}, \mathbf{C}_{\hat{u}-1}, \mathbf{S}_{\hat{u}-1} \,\rangle \;\rightarrow_{i:(\text{RCV-ACC-REQ})} \langle\, \mathbf{B}_{\hat{u}}, \mathbf{C}_{\hat{u}}, \mathbf{S}_{\hat{u}} \,\rangle$$

with $\mathbf{S}_{\hat{u}}(i) = \big(\cdot, \cdot, \hat{r}, (\hat{v}, \hat{r}), (\top, \cdot), \cdot\big)$. By Lemma 9.1.2.1 $t < \hat{u} < u$. By Lemma 9.1.2.7(2) we have $s' \le \hat{r}$ and $\hat{r} \le s$, so $\hat{r} = s' = s < r'$. We have reached a contradiction, so the hypothesis $\mathbf{C}_z.\mathsf{P3}^{\hat{r}}_{i\to\text{lead}(\hat{r})} \neq \bot$ must be false.

2. If $s' < s$, suppose by contradiction that, for all $\hat{r}$, $r' \le \hat{r} < r$ it holds $\mathbf{C}_z.\mathsf{P3}^{\hat{r}}_{i\to\text{lead}(\hat{r})} = \bot$. Then, by applying Lemma 9.1.2.10(1) to all request numbers $\hat{r}$ with $\hat{r}$, $r' \le \hat{r} < r$, for which $\mathbf{C}_z.\mathsf{P1}^{\hat{r}}_{i\to\text{lead}(\hat{r})} \neq \bot$ we derive $s' = s$. We have reached a contradiction, so the hypothesis $\mathbf{C}_z.\mathsf{P3}^{\hat{r}}_{i\to\text{lead}(\hat{r})} = \bot$ for all $\hat{r}$, $r' \le \hat{r} < r$, must be false.

□

The following lemma states that the proposal (v, s) proposed by some process i in a certain round r originates precisely from round s.

Lemma 9.1.2.12 (Proposal origin) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)^{(v_1,\ldots,v_n)}_{T}$ *be a run and* $z \in T$. *If* $\mathbf{C}_z.\mathsf{P1}^{r}_{i\to\text{lead}(r)} = ((v, s), \cdot)$, *with* $r > 1$ *and* $s > 0$, *then* $\mathbf{C}_z.\mathsf{P3}^{s}_{i\to\text{lead}(s)} = (v, \cdot)$ *and* $v = \text{val}^s(\mathbf{C}_z)$.

Proof. Suppose by contradiction that $\mathbf{C}_z.\mathsf{P3}^{s}_{i\to\text{lead}(s)} = \bot$. Suppose $\{\; r' \in \text{acc-prep}_i(R) \mid r' \le s \;\} \neq \emptyset$ and take $\hat{r} = \max\{\; r' \in \text{acc-prep}_i(R) \mid r' \le s \;\}$. Note that if $s \in \text{acc-prep}_i(R)$ then $\hat{r} = s$, otherwise $\hat{r} < s$. By definition of $\text{acc-prep}_i(R)$ and Corollary 9.1.2.5(1) $\mathbf{C}_z.\mathsf{P1}^{\hat{r}}_{i\to\text{lead}(\hat{r})} \neq \bot$. Let us say that $\mathbf{C}_z.\mathsf{P1}^{\hat{r}}_{i\to\text{lead}(\hat{r})} = ((\hat{v}, \hat{s}), \cdot)$. From Lemma 9.1.2.8(1), we know that $\hat{s} < \hat{r} \le s < r$. Let us now consider $\mathbf{C}_z.\mathsf{P1}^{\text{f}^{\hat{r}}_i(R)}_{i\to\text{lead}(\text{f}^{\hat{r}}_i(R))}$. By definition of $\text{f}^{\hat{r}}_i(R)$ and $\hat{r}$, we have $s < \text{f}^{\hat{r}}_i(R)$. Since $\mathbf{C}_z.\mathsf{P3}^{s}_{i\to\text{lead}(s)} = \bot$, if $\hat{r} = s$ by Lemma 9.1.2.10(1) we have that $\mathbf{C}_z.\mathsf{P1}^{\text{f}^{\hat{r}}_i(R)}_{i\to\text{lead}(\text{f}^{\hat{r}}_i(R))} = ((\hat{v}, \hat{s}), \cdot)$; if instead $\hat{r} < s$ by Lemma 9.1.2.10 we can have either $\mathbf{C}_z.\mathsf{P1}^{\text{f}^{\hat{r}}_i(R)}_{i\to\text{lead}(\text{f}^{\hat{r}}_i(R))} = ((\hat{v}, \hat{s}), \cdot)$ (if $\mathbf{C}_z.\mathsf{P3}^{\hat{r}}_{i\to\text{lead}(\hat{r})} = \bot$) or $\mathbf{C}_z.\mathsf{P1}^{\text{f}^{\hat{r}}_i(R)}_{i\to\text{lead}(\text{f}^{\hat{r}}_i(R))} = ((\cdot, \hat{r}), \cdot)$ (if $\mathbf{C}_z.\mathsf{P3}^{\hat{r}}_{i\to\text{lead}(\hat{r})} \neq \bot$). Since, in any case, the proposal stamp ($\hat{s}$, $\hat{s}$ and $\hat{r}$ respectively) is strictly smaller than s, we know that $\text{f}^{\hat{r}}_i(R)$ cannot be equal to r. And by Lemma 9.1.2.8(2) we know that $\text{f}^{\hat{r}}_i(R) < r$.
Moreover, by Lemma 9.1.2.11(2), there must be (at least) a proposal number $\tilde{r}$, with $\text{f}^{\hat{r}}_i(R) \le \tilde{r} < r$ such that $\mathbf{C}_z.\mathsf{P3}^{\tilde{r}}_{i\to\text{lead}(\tilde{r})} = (\tilde{v}, \cdot)$. So, from Lemma 9.1.2.10(2), we have $\mathbf{C}_z.\mathsf{P1}^{\text{f}^{\tilde{r}}_i(R)}_{i\to\text{lead}(\text{f}^{\tilde{r}}_i(R))} = ((\tilde{v}, \tilde{r}), \cdot)$ with $\text{f}^{\tilde{r}}_i(R) \le r$.

From Lemma 9.1.2.8(2) we have $\tilde{r} \leq s$, which is not the case since $s < \mathrm{f}_i^{\hat{r}}(R) \leq$. We have reached a contradiction, so the hypothesis $\mathbf{C}_z.\mathsf{P3}^s_{i\to\mathrm{lead}(s)} = \bot$ must be false. By Lemma 9.1.2.10(2) and $\mathbf{C}_z.\mathsf{P3}^s_{i\to\mathrm{lead}(s)} \neq \bot$, we get $\mathbf{C}_z.\mathsf{P1}^{\mathrm{f}_i^s(R)}_{i\to\mathrm{lead}(\mathrm{f}_i^s(R))} = ((\mathrm{val}^s(\mathbf{C}_z), s), \cdot)$. This together with Lemma 9.1.2.11(1) allows us to derive $v = \mathrm{val}^s(\mathbf{C}_z)$. Suppose now $\{\ r' \in \text{acc-prep}_i(R) \mid r' \leq s\ \}=\emptyset$. Since $\mathbf{C}_z.\mathsf{P3}^s_{i\to\mathrm{lead}(s)}=\bot$, it must be $\mathbf{C}_z.\mathsf{P1}^{\mathrm{f}_i^s(R)}_{i\to\mathrm{lead}(\mathrm{f}_i^s(R))} = ((\cdot, 0), \cdot)$, with $0 < s$, so the reasoning goes exactly as before. □

9.2 Consensus Properties

In this section, we prove two of the three main Consensus properties: *Validity* and *Agreement*.

9.2.1 Validity

Intuitively, in a generic run, whenever a value appears as an estimate, a proposal, or a decision —i.e., within one of the components $\mathbf{B}$, $\mathbf{C}$ or $\mathbf{S}$— then this value is one of the initially proposed values. In other words, values are never invented. The Validity property of the Consensus algorithm coincides with the last item of the following theorem.

Theorem 9.2.1.1 (Validity) *Let $\big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)^{(v_1,\ldots,v_n)}_T$ be a run. Then, for all $t \in T$, the conjunction of the following properties holds:*

1. *If $\mathbf{B}_t(r) = (v, \cdot)$, for some $0 < r$, then there is $k \in \mathbb{P}$ with $v = v_k$.*
2. *If $\mathbf{C}_t.\mathsf{P1}^r_{i\to\mathrm{lead}(r)} = ((v, \cdot), \cdot)$, for some $i \in \mathbb{P}$ and $0 < r$, then there is $k \in \mathbb{P}$ with $v = v_k$.*
3. *If $\mathbf{C}_t.\mathsf{P2}^r_{\mathrm{lead}(r)\to i} = ((v, \cdot), \cdot)$, for some $i \in \mathbb{P}$ and $0 < r$, then there is $k \in \mathbb{P}$ with $v = v_k$.*
4. *If $\mathbf{S}_t(i) = \big(\cdot, (v, \cdot), \cdot, \cdot\big)$, for some $i \in \mathbb{P}$, then there is $k \in \mathbb{P}$ with $v = v_k$.*
5. *If $\mathbf{S}_t(i) = \big(\cdot, \cdot, \cdot, (v, \cdot)\big)$, for some $i \in \mathbb{P}$, then there is $k \in \mathbb{P}$ with $v = v_k$.*

Proof. By induction on the length t of the (prefix of a) run

$$\langle\, \mathbf{B}_0, \mathbf{C}_0, \mathbf{S}_0^{(v_1,\ldots,v_n)} \,\rangle \ \rightarrow^t \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle.$$

Base case $t = 0$: Immediate since $\mathbf{B}_0 = \emptyset$, $\mathbf{C}_0 = \emptyset$ and $\mathbf{S}(i) = \mathbf{S}_0^{(v_1,\ldots,v_n)} = \big((\mathrm{no}, \mathrm{idle}), 0, 0, (v_i, 0), (\top, 1), \bot\big)$ for $i \in \mathbb{P}$.

Inductive case $t > 0$ **:** We prove the step from $t{-}1$ to t (if $t \in T$) referring to the step

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \ \rightarrow_{j:(\textsc{Rule})} \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle \tag{9.1}$$

By exhaustive analysis, we check for every rule (RULE) that each of the five conditions is satisfied in configuration $\langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle$. All cases are similar: in every possible step, values that appear in the conclusions of the applied rule are just copied via the premises referring to the former configuration, such that the inductive hypothesis applies and proves the required origin in the initial configuration.

case $(\textsc{Rule}) = (\textsc{snd-acc-req})$ for some $0 < r$:

1. $\mathbf{B}_t = \mathbf{B}_{t-1}$, so the inductive hypothesis is already sufficient.
2. $\mathbf{C}_t.\mathsf{P1}^r = \mathbf{C}_{t-1}.\mathsf{P1}^r$ for all $0 < r$, so the inductive hypothesis is already sufficient.
3. $\mathbf{C}_t = \mathbf{C}_{t-1}\{\, \mathsf{P2}^r_{\mathrm{lead}(r)\to j} \mapsto (\mathrm{best}^r(\mathbf{C}_{t-1}), (1,0,0,0))\,\}_{\forall j \in \mathbb{P}}$, so in addition to the inductive hypothesis on $\mathbf{C}_{t-1}$, we only need to show that $\mathrm{best}^r(\mathbf{C}_{t-1}) = v_k$ for some $k \in \mathbb{P}$. This is guaranteed by the projective definition of $\mathrm{best}^r(\cdot)$ together with the inductive hypothesis on $\mathbf{C}_{t-1}$.
4. $\mathbf{S}_t(j) = ((\mathrm{yes}, \mathrm{polling}), r, p, (v, r), (\top, \cdot), d)$, since the only change in the state of i is in its phase field (which passes from waiting to polling), and since $\mathbf{S}_t(i) = \mathbf{S}_{t-1}(i)$ for all $i \neq j$, the inductive hypothesis is already sufficient.
5. Completely analogous to the previous case, except arguing with d instead of b.

case $(\textsc{Rule}) \neq (\textsc{snd-acc-req})$: Analogous to the previous, just that there is not even an indirection referring to an externally defined function like $\mathrm{best}^r(\cdot)$.

□

9.2.2 Agreement

In our terminology, a value v is *locked* for round r (in a message history $\mathbf{C}$) if enough 3rd-phase messages are defined for request number r; the transmission state of these messages is not important.

Definition 9.2.2.1 *A value v is called* locked *for request number r in a* CH-*message history $\mathbf{C}$, written $\mathbf{Q} \overset{r}{\looparrowright} v$, if $v = \mathrm{val}^r(\mathbf{C})$ and $|\,\mathrm{ack}_v(\mathbf{C}.\mathsf{P3}^r)| \geq \lceil (n{+}1)/2 \rceil$.*

Notice the convenience of having at hand the history abstraction to access the messages that were sent in the past, without having to look at the run leading to the current state.

We now show that whenever a RBC-broadcast occurs, it is for a locked value.

Lemma 9.2.2.2 (Locking) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run where* $\mathbf{B}_z(r) = (v, \cdot)$ *for some* $z \in T$, $0 < r \in \mathbb{N}$, $v \in \mathbb{V}$. *Then* $\mathbf{Q}_z \overset{r}{\looparrowright} v$.

Proof. Since $\mathbf{B}_0(r) = \bot$ and $\mathbf{B}_z(r) \neq \bot$, by Lemma 9.1.2.2(5), there exists $t \in T$, with $t \leq z$, for which

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \ \rightarrow_{\text{lead}(r):(\text{SUCCESS})} \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle$$

such that

$$\begin{array}{rclcrcl} \mathbf{B}_{t-1}(r) & = & \bot & \quad & \mathbf{S}_{t-1}(\text{lead}(r)) & = & \big((\text{yes}, \text{polling}), r, \cdot, \cdot, \cdot, \cdot\big) \\ \mathbf{B}_t(r) & = & (v, \mathbb{P}) & & \mathbf{S}_t(\text{lead}(r)) & = & \big((\text{yes}, \text{done}), r, \cdot, \cdot, \cdot, \cdot\big)\ . \end{array}$$

As one of the premises of rule (SUCCESS) is the existence of $\mathbf{C}_{t-1}.\mathsf{P3}^r_{j\to\text{lead}(r)} = (v, \cdot)$, there exist a $j \in \mathbb{P}$ and a $u \in T$, $u < t$ (notice that it cannot be $u = t$), for which

$$\langle\, \mathbf{B}_{u-1}, \mathbf{C}_{u-1}, \mathbf{S}_{u-1} \,\rangle \ \rightarrow_{j:(\text{RCV-ACC-REQ})} \langle\, \mathbf{B}_u, \mathbf{C}_u, \mathbf{S}_u \,\rangle$$

such that $\mathbf{S}_{u-1}(j) = \big(\cdot, \cdot, p, (v_j, s_j), \cdot, \cdot\big)$ and $\mathbf{S}_u(i) = \big(\cdot, \cdot, p, (v, p), \cdot, \cdot\big)$. As one of the premises of rule (RCV-ACC-REQ) is $\mathbf{C}_{u-1}.\mathsf{P2}^p_{\text{lead}(p)\to j} = ((v, s), (\cdot, \cdot, \cdot, \delta))$ with $\delta \geq 1$, by Definition 7.2.0.2 we have $v = \text{val}^r(\mathbf{C}_{u-1})$.

By definition, we have $|\,\text{ack}_v(\mathbf{C}_{t-1}).\mathsf{P3}^r| \geq |\,\text{received}(\mathbf{C}_{t-1}).\mathsf{P3}^r_\iota|$. By Corollary 9.1.2.5(1), we obtain $|\,\text{ack}_v(\mathbf{C}_z).\mathsf{P3}^r| \geq |\,\text{received}(\mathbf{C}_z).\mathsf{P3}^r_\iota|$. By Corollary 9.1.2.6, we obtain $\text{val}^r(\mathbf{C}_{u-1}) = \text{val}^r(\mathbf{C}_z)$ This, together with the premise $|\,\text{received}(\mathbf{C}_{t-1}).\mathsf{P3}^r_\iota| \geq \lceil (n{+}1)/2 \rceil$ of rule (SUCCESS) give us $\mathbf{Q}_z \overset{r}{\looparrowright} v$.

□

Now, we can move to look at agreement properties. The key idea is to prove that lockings in two different request numbers must be for the very same value.

Proposition 9.2.2.3 (Locking agreement) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run. If there are a state* $z \in T$, *request number numbers* $r_1, r_2 \in \mathbb{N}$ *and values* $v_1, v_2 \in \mathbb{V}$ *such that* $\mathbf{Q}_z \overset{r_1}{\looparrowright} v_1$ *and* $\mathbf{Q}_z \overset{r_2}{\looparrowright} v_2$, *then* $v_1 = v_2$.

Proof. Let us assume $r_1 \leq r_2$ (the proof for the opposite case is analogous). From Definition 9.2.2.1, we have $v_1 = \mathrm{val}^{r_1}(\mathbf{C}_z)$ and $v_2 = \mathrm{val}^{r_2}(\mathbf{C}_z)$. We will prove that $\mathrm{val}^{r_1}(\mathbf{C}_z) = \mathrm{val}^{r_2}(\mathbf{C}_z)$ by strong induction on r_2 starting from r_1.

Base case If $r_1 = r_2$ then the result is trivially true.
Inductive case Let $r_1 < r_2$.

By inductive hypothesis we assume that for all $r \in \mathbb{N}$, $r_1 \leq r \leq r_2{-}1$, for which $\mathrm{val}^{r}(\mathbf{C}_z) \neq \bot$ we have

$$\mathrm{val}^{r}(\mathbf{C}_z) = \mathrm{val}^{r_1}(\mathbf{C}_z) = v_1.$$

By definition of $\mathrm{val}^{r_2}(\mathbf{C}_z)$ (Definition 7.2.0.2), we have $\mathbf{C}_z.\mathsf{P2}^{r_2}_{\mathrm{lead}(r_2)\to i} = ((\mathrm{val}^{r_2}(\mathbf{C}_z), \cdot), \cdot)$, for all $i \in \mathbb{P}$. By Lemma 9.1.2.2(3), there is $t \in T$, $t \leq z$, for which

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \ \rightarrow_{\mathrm{lead}(r_2):(\textsc{snd-acc-req})} \langle\, \mathbf{B}_{t}, \mathbf{C}_{t}, \mathbf{S}_{t} \,\rangle$$

such that, for all $i \in \mathbb{P}$,

$$\begin{array}{rcl} \mathbf{C}_{t-1}.\mathsf{P2}^{r_2}_{\mathrm{lead}(r_2)\to i} & = & \bot \\ \mathbf{C}_{t}.\mathsf{P2}^{r_2}_{\mathrm{lead}(r_2)\to i} & = & (\mathrm{best}^{r_2}(\mathbf{C}_{t-1}), \cdot) \end{array}$$

From Table 7.2, we see that one of the premises of rule (SND-ACC-REQ) is $|\,\mathrm{received}(\mathbf{C}_{t-1}).\mathsf{P1}^{r_2}_{\iota}| \geq \lceil (n{+}1)/2 \rceil$.
Let $C_{r_2} = \{\, j \mid \mathbf{C}_z.\mathsf{P1}^{r_2}_{j\to\mathrm{lead}(r_2)} = (\cdot, (\cdot, \cdot, \cdot, \delta))$ with $\delta \geq 1\,\}$, then by Corollary 9.1.2.5(1) on the monotonicity of message histories we have $|\, C_{r_2}\,| \geq |\,\mathrm{received}(\mathbf{C}_{t-1}).\mathsf{P1}^{r_2}_{\iota}\,| \geq \lceil (n{+}1)/2 \rceil$.
Let $L_{r_1} = \{\, j \mid \mathbf{C}_z.\mathsf{P3}^{r_1}_{j\to\mathrm{lead}(r_1)} = (v_1, \cdot)\,\}$. As $\mathbf{Q}_z \overset{r_1}{\looparrowright} v_1$, by Definition 9.2.2.1 we have $|\,\mathrm{ack}_v(\mathbf{C}).\mathsf{P3}^{r}| \geq \lceil (n{+}1)/2 \rceil$ and hence $|\, L_{r_1}\,| \geq \lceil (n{+}1)/2 \rceil$.
Since the cardinality of both C_{r_2} and L_{r_1} is greater than $\lceil (n{+}1)/2 \rceil$, and $C_{r_2}, L_{r_1} \subseteq \{1, \ldots, n\}$, it follows that $L_{r_1} \cap C_{r_2} \neq \emptyset$. Let us take a process $k \in L_{r_1} \cap C_{r_2}$. As $k \in C_{r_2}$, we have $\mathbf{C}_z.\mathsf{P1}^{r_2}_{k\to\mathrm{lead}(r_2)} \neq \bot$.
Since $k \in L_{r_1}$, we have $\mathbf{C}_z.\mathsf{P3}^{r_1}_{k\to\mathrm{lead}(r_1)} \neq \bot$. By Lemma 9.1.2.10(2)

$$\mathbf{C}_z.\mathsf{P1}^{\mathrm{f}^{r_1}_i(R)}_{k\to\mathrm{lead}(\mathrm{f}^{r_1}_i(R))} = \big(\big(\,\mathrm{val}^{r_1}(\mathbf{C}_z), r_1\big), \cdot\big)\,.$$

There are two cases: $\mathrm{f}^{r_1}_i(R) < r_2$ and $\mathrm{f}^{r_1}_i(R) = r_2$. In the former case, since $k \in C_{r_2}$, we have

$$\mathbf{C}_z.\mathsf{P1}^{r_2}_{k\to\mathrm{lead}(r_2)} = \big((\cdot, \tilde{r}), (\cdot, \cdot, \cdot, \delta)\big) \ \text{ with } \ \delta \geq 1$$

for some $\tilde{r} \in \mathbb{N}$. By Lemma 9.1.2.8(1), $\tilde{r} < r_2$. As $\mathrm{f}_i^{r_1}(R) < r_2$, by Lemma 9.1.2.8(2) we have $r_1 \leq \tilde{r}$. In the latter case, we have directly

$$\mathbf{C}_z.\mathsf{P1}^{r_2}_{k\rightarrow\mathrm{lead}(r_2)} = \big(\big(\,\mathrm{val}^{r_1}(\mathbf{C}_z), r_1\big), \cdot\big)$$

and the $\tilde{r}$ of the previous case coincides with r_1.

Let us now consider the process $i \in C_{r_2}$ that has proposed the best value in request number r_2, according to Definition 7.2.0.1. Then by Definition 7.2.0.2 we have

$$\mathbf{C}_z.\mathsf{P1}^{r_2}_{i\rightarrow\mathrm{lead}(r_2)} = \big(\big(\,\mathrm{val}^{r_2}(\mathbf{C}_z), \hat{r}\big), (\cdot,\cdot,\cdot,\delta)\big) \;\text{ with }\; \delta \geq 1$$

for some $\hat{r} \in \mathbb{N}$. By applying Lemma 9.1.2.8(1), we get $\hat{r} < r_2$. As i, in request number r_2, has proposed the value with the best timestamp $\hat{r}$, it follows that $\tilde{r} \leq \hat{r}$.

Putting it altogether, we get $r_1 \leq \tilde{r} \leq \hat{r} \leq r_2{-}1$. By the inductive hypothesis, $\mathrm{val}^{\hat{r}}(\mathbf{C}_z) = \mathrm{val}^{r_1}(\mathbf{C}_z)$.

From $\mathbf{C}_z.\mathsf{P1}^{r_2}_{i\rightarrow\mathrm{lead}(r_2)} = \big(\big(\,\mathrm{val}^{r_2}(\mathbf{C}_z), \hat{r}\big), (\cdot,\cdot,\cdot,\delta)\big)$ with $\delta \geq 1$ and Lemma 9.1.2.12, we get that $\mathrm{val}^{\hat{r}}(\mathbf{C}_z) = \mathrm{val}^{r_2}(\mathbf{C}_z)$. Thus, $\mathrm{val}^{r_1}(\mathbf{C}_z) = \mathrm{val}^{r_2}(\mathbf{C}_z)$, as required.

$\square$

Now, everything is in place to prove the property of Agreement: Whenever two processes i and j have decided, as expressed within the respective state components, they have done so for the same value.

Theorem 9.2.2.4 (Agreement) *Let* $\big(\langle\, \mathbf{B}_x, \mathbf{C}_x, \mathbf{S}_x \,\rangle\big)^{(v_1,\ldots,v_n)}_T$ *be a run. If there are a time* $z \in T$*, processes* $i, j \in \mathbb{P}$*, and values* $v_1, v_2 \in \mathbb{V}$ *such that* $\mathbf{S}_z(i) = (\cdot,\cdot,\cdot,(v_1,\cdot))$ *and* $\mathbf{S}_z(j) = (\cdot,\cdot,\cdot,(v_2,\cdot))$*, then* $v_1 = v_2$*.*

Proof. In the initial state $\mathbf{S}_0$ the decision component of both process i and process j is undefined (see Section 7.1), while in state $\mathbf{S}_z$ it is defined. By inspection of the rules of Table 7.2, the decision element of a process state can be changed only once per run, by applying rule (RBC-DELIVER). In particular, it can change from $\perp$ to something of the form $(\hat{v}, \hat{r})$ executing rule (RBC-DELIVER) for the first time. When executing rule (RBC-DELIVER) the following times, the decision value is left unchanged. Therefore, once initialized, the decision value of a process state stays the same until the end of the run. As a consequence, there are $t \in T$ and $u \in T$, with $t \leq z$ and $u \leq z$, such that

$$\langle\, \mathbf{B}_{t-1}, \mathbf{C}_{t-1}, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{i:(\textsc{Rbc-deliver})}\; \langle\, \mathbf{B}_t, \mathbf{C}_t, \mathbf{S}_t \,\rangle$$

with $\mathbf{S}_{t-1}(i) = (\cdot,\cdot,\cdot,\cdot,\cdot,\bot)$, $\mathbf{S}_t(i) = (\cdot,\cdot,\cdot,\cdot,\cdot,(v_1,r_1))$, $\mathbf{B}_{t-1}(r_1) = (v_1,\cdot)$ for some $r_1 > 0$, and

$$\langle\, \mathbf{B}_{u-1}, \mathbf{C}_{u-1}, \mathbf{S}_{u-1} \,\rangle \;\rightarrow_{j:(\textsc{Rbc-deliver})} \langle\, \mathbf{B}_u, \mathbf{C}_u, \mathbf{S}_u \,\rangle$$

with $\mathbf{S}_{u-1}(j) = (\cdot,\cdot,\cdot,\cdot,\cdot,\bot)$, $\mathbf{S}_u(i) = (\cdot,\cdot,\cdot,\cdot,\cdot,(v_2,r_2))$, $\mathbf{B}_{u-1}(r_2) = (v_2,\cdot)$ for some $r_2 > 0$.

As $u, t \leq z$, by Corollary 9.1.2.5(2), $\mathbf{B}_z(r_1) = (v_1,\cdot)$ and $\mathbf{B}_z(r_2) = (v_2,\cdot)$.

By Lemma 9.2.2.2, it follows that $\mathbf{Q}_z \overset{r_1}{\looparrowright} v_1$ and $\mathbf{Q}_z \overset{r_2}{\looparrowright} v_2$.

By Proposition 9.2.2.3, we conclude $v_1 = v_2$.

□

Chapter 10

The Paxos Algorithm Does Not Terminate

Whether Paxos satisfies the Termination property or not is a controversial issue and we feel that researchers have not yet ”reached consensus” on the matter. Some say that Lamport has actually never claimed that his algorithm gives the certainty of terminating. But in this case our complaint would be that Paxos cannot be considered a Consensus algorithm since, as far as we understand, an algorithm that solves the Consensus problem has to guarantee that all the three properties of Validity, Agreement and Termination are respected [1]. Besides, in [36] Lamport states that *”[i]f enough of the system (proposer, acceptors, and communication network) is working properly, liveness can therefore be achieved by electing a single distinguished proposer”*, which makes us believe that Lamport took care of the issue of Termination and he knew under which conditions the algorithm terminates.

Unfortunately, such conditions have never been clearly spelled out (can anyone quantify how much is *enough*?) and many still believe that the simple fact of requiring the eventual presence of one and only one leader is enough to ensure Termination. This is not true, and in this section we will show, using our formalism, that even in presence of perfectly favorable conditions (no crashes, no message losses, a unique leader during the whole run) the algorithm as it is might not terminate.

Intuitively, this is due to the fact that, in [35], Lamport defines the action *”Try New Ballot”* as being *”always enabled”*. This absence of conditions on the beginning of a new ballot is necessary to avoid blocking for the fol-

[1]Note also that, if the Termination property was not required from an algorithm that solves the Consensus problem, then the FLP result would have no reason of existing.

lowing three reasons: the fist one is that messages can get lost and a leader might never get what it is waiting for; the second one is that processes can have already promised another leader their participation to a greater round, and they would not reply to requests for a lower round; the third reason is that processes can crash and they can be in the impossibility of replying to a leader, moreover, since they keep a very limited memory of the past, recovered processes ignore (and therefore do not answer to) messages received before or during the period when they were crashed. So, having *"Try New Ballot"* always enabled constitutes the strength of Paxos with respect to message losses, number of messages in transit and process faults, but it constitutes also its weak link from the termination point of view because it can allow a "stupid" leader to start a new ballot after the other without waiting for long enough to receive potential messages from other processes. A leader can be "stupid" for multiple reasons, and such reasons do not necessarily have to be malicious. For example, we can consider a leader to be "stupid" when it does not know the time it takes for a message to reach the destination process, be processed and be sent back. But this lack of knowledge is absolutely normal in an asynchronous system, and it is actually part of the game!

So, here is a summary of the conditions that both Lamport [35] and De Prisco, Lampson, Lynch [17] present in their papers without putting too much relevance on them: eventually the system must be synchronous, the leader must have knowledge of the interval of time that it takes for messages to do a round trip and to be processed by other processes, and finally there must be a period of time (corresponding to the mentioned interval) in which there is a majority of processes that is not crashed. These, to our eyes, are not trivial or negligible conditions and he who implements the algorithm should be fully aware of them. In fact, one cannot write the algorithm independently of the system in which it will run, and even a little change in the system (for example a slow down of a process [2]) can compromise the termination of the algorithm.

10.1 As It Is, Paxos Does Not Terminate

We now use our formalism to show that, even if we restrict ourselves to a system where there are no crashes, where there is a unique leader and

[2]Imagine a system administrator that, in order to perform maintenance operations, substitutes for a couple of hours a computer of the original system with an older and slower one.

where channels are completely reliable, we can always find a run that is not satisfying the Termination property.

But let us first define the properties of the system. Even though Paxos is designed to work in crash-recovery failure model systems, we strengthen our correctness definition by saying that a process is *correct* in a given run, if it does never crash in that run. To formalize this notion in our model, we just need to check the definedness of the relevant state entry in all configurations of the given run.

Definition 10.1.0.5 (Correct Processes) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$*. We define*

$$\mathsf{Correct}(R) := \{\, i \in \mathbb{P} \mid \forall t \in T : \mathbf{S}_t(i) \neq (\cdot,\cdot,\cdot,\cdot,(\bot,\cdot),\cdot) \,\}\,.$$

A process $i \in \mathsf{Correct}(R)$ *is called* correct in R*.*

To formally state the complete reliability property of the messaging service based on our model, we need forbid the loss and the replication of the messages and we have to impose that for each sending of a message there is a reception of it.

Definition 10.1.0.6 (Completely Reliable Point-To-Point) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *denote a run.*
R *satisfies* Complete-CH-Reliability *if:*

1. *there is no* $t \in T$ *such that*

$$\langle\, \cdot, \mathbf{Q}_{t-1}, \cdot \,\rangle \;\rightarrow_{(\mathrm{R})}\; \langle\, \cdot, \mathbf{Q}_t, \cdot \,\rangle$$

with (R) *equals to* (CH_DUPL) *or* (CH_LOSS)*;*

2. *for all processes* $k \in \mathsf{Correct}(R)$ *and time* $t > 0$*, whenever*

$$\langle\, \cdot, \mathbf{Q}_{t-1}, \cdot \,\rangle \;\rightarrow_{j:(\cdot)}\; \langle\, \cdot, \mathbf{Q}_t, \cdot \,\rangle$$

with $\mathbf{Q}_{t-1}.P^r_{j\rightarrow k} = \bot$ *and* $\mathbf{Q}_t.P^r_{j\rightarrow k} = (\cdot,(1,0,0,0))$*, for* $r > 0$ *and* $P \in \{\, \mathsf{P0}, \mathsf{P1}, \mathsf{P2}, \mathsf{P3} \,\}$*, then there is* $u > t$ *such that*

$$\langle\, \cdot, \mathbf{Q}_{u-1}, \cdot \,\rangle \;\rightarrow_{k:(\text{CH-DELIVER})}\; \langle\, \cdot, \mathbf{Q}_u, \cdot \,\rangle$$

with $\mathbf{Q}_u.P^r_{j\rightarrow k} = (\cdot,(0,\cdot,\cdot,\delta))$ *and* $\delta \geq 1$*.*

Now, we formalize the property of Unique Leader in our rewrite system by expressing the fact that one and only one process can become leader in a run, and when it does it stays so forever.

Definition 10.1.0.7 (Unique Leader) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *be a run. R satisfies the property of Unique Leader if there is one and only one $t \in T$ such that*

$$\langle\, \cdot, \cdot, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{i:(\textsc{leader-yes})}\; \langle\, \cdot, \cdot, \mathbf{S}_t \,\rangle$$

with $i \in \mathsf{Correct}(R)$, and there is no $t \in T$ such that

$$\langle\, \cdot, \cdot, \mathbf{S}_{t-1} \,\rangle \;\rightarrow_{i:(\textsc{leader-no})}\; \langle\, \cdot, \cdot, \mathbf{S}_t \,\rangle$$

for any $i \in \mathbb{P}$.

Definition 10.1.0.8 (Reliable Broadcast) *Let* $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ *denote a run. Then we define some of its properties:*

1. RBC-Validity: *for every $i \in \mathsf{Correct}(R)$, if there is $t > 0$ such that*

   $$\langle\, \mathbf{B}_{t-1}, \cdot, \cdot \,\rangle \;\rightarrow_{i:(\textsc{success})}\; \langle\, \mathbf{B}_t, \cdot, \cdot \,\rangle$$

   with $\mathbf{B}_{t-1}(r) = \bot$ and $\mathbf{B}_t(r) = (v, \mathbb{P})$ for some $r > 0$, then there is $u > t$ such that

   $$\langle\, \mathbf{B}_{u-1}, \cdot, \cdot \,\rangle \;\rightarrow_{i:(\textsc{Rbc-deliver})}\; \langle\, \mathbf{B}_u, \cdot, \cdot \,\rangle$$

   with $\mathbf{B}_{u-1}(r) = (v, D_{u-1})$, $\mathbf{B}_u(r) = (v, D_u)$ and $D_{u-1} = D_u \uplus \{i\}$.
2. RBC-Agreement: *for every $i \in \mathsf{Correct}(R)$, if there is $t > 0$ such that*

   $$\langle\, \mathbf{B}_{t-1}, \cdot, \cdot \,\rangle \;\rightarrow_{i:(\textsc{Rbc-deliver})}\; \langle\, \mathbf{B}_t, \cdot, \cdot \,\rangle$$

   with $\mathbf{B}_{t-1}(r) = (v, D_{t-1})$, $\mathbf{B}_t(r) = (v, D_t)$ and $D_{t-1} = D_t \uplus \{i\}$ for some $r > 0$,
   then, for every $j \in \mathsf{Correct}(R)$, there is $u > 0$ such that

   $$\langle\, \mathbf{B}_{u-1}, \cdot, \cdot \,\rangle \;\rightarrow_{j:(\textsc{Rbc-deliver})}\; \langle\, \mathbf{B}_u, \cdot, \cdot \,\rangle$$

   with $\mathbf{B}_{u-1}(r) = (v, D_{u-1})$, $\mathbf{B}_u(r) = (v, D_u)$ and $D_{u-1} = D_u \uplus \{j\}$.
3. (Uniform) RBC-Integrity: *for every $i \in \mathsf{Correct}(R)$, if there are $t_1, t_2 > 0$ such that*

   $$\langle\, \mathbf{B}_{t_1-1}, \cdot, \cdot \,\rangle \;\rightarrow_{i:(\textsc{Rbc-deliver})}\; \langle\, \mathbf{B}_{t_1}, \cdot, \cdot \,\rangle$$

   $$\langle\, \mathbf{B}_{t_2-1}, \cdot, \cdot \,\rangle \;\rightarrow_{i:(\textsc{Rbc-deliver})}\; \langle\, \mathbf{B}_{t_2}, \cdot, \cdot \,\rangle$$

   and, for some $r > 0$, both

   (a) $\mathbf{B}_{t_1-1}(r) = (v, D_{t_1-1})$, $\mathbf{B}_{t_1}(r) = (v, D_{t_1})$, *and* $D_{t_1-1} = D_{t_1} \uplus \{i\}$.
   (b) $\mathbf{B}_{t_2-1}(r) = (v, D_{t_2-1})$, $\mathbf{B}_{t_2}(r) = (v, D_{t_2})$, *and* $D_{t_2-1} = D_{t_2} \uplus \{i\}$.

then $t_1 = t_2$ and there is $u < t_1$ such that

$$\langle\, \mathbf{B}_{u-1}, \cdot, \cdot \,\rangle \ \rightarrow_{i:(\text{SUCCESS})} \langle\, \mathbf{B}_{u}, \cdot, \cdot \,\rangle$$

with $\mathbf{B}_{u-1}(r) = \bot$ and $\mathbf{B}_u(r) = (v, \mathbb{P})$.

Finally, we put all the properties together to select those runs of Consensus where no process may crash, where there is a unique leader and where the communication services are perfectly working.

Definition 10.1.0.9 (Perfect Run) *Let $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ be a run. We say that R is a* Perfect Run *if it satisfies the property of Unique Leader, the property of Complete-*CH*-Reliability, the three properties of Reliable Broadcast and if all the processes are correct, i.e.* $\mathsf{Correct}(R) = \mathbb{P}$.

Now everything is in place to show that even perfect runs might not terminate.

Theorem 10.1.0.10 (Non Termination) *Let $R = \big(\langle\, \mathbf{B}_x, \mathbf{Q}_x, \mathbf{S}_x \,\rangle\big)_T^{(v_1,\ldots,v_n)}$ be a Perfect Run. R can be infinite.*

Proof. From Definition 10.1.0.7 we know that the unique leader of R belongs to the set $\mathsf{Correct}(R)$. Thus, such leader can never crash in R. If we examine rule (NEW), we notice that it is always enabled for any non-crashed leader that has not yet decided. Thus, the unique leader of R can execute rule (NEW) at any time in R, provided that it has not decided. Therefore, if we take the run

$$\langle\, \mathbf{B}_0, \mathbf{C}_0, \mathbf{S}_0^{(v_1\ldots,v_n)} \,\rangle \ \rightarrow_{i:(\text{LEADER-YES})} \langle\, \mathbf{B}_1, \mathbf{C}_1, \mathbf{S}_1 \,\rangle \ \rightarrow^{\infty}_{i:(\text{NEW})}$$

we have found a Perfect Run that is infinite. □

Putting ourselves in the best conditions we can imagine for a system and showing that, even under these excellent conditions, Paxos might not terminate allows us to argue that the Termination problem is an intrinsic problem of the Paxos algorithm, that does not depend on the characteristics of the underlying layers.

We agree with the reader that an infinite run like the one presented in the proof above is highly improbable, but the simple fact that such a run exists should alert us to the possibility of non termination of the algorithm. Moreover, there can be infinite other runs where the algorithm evolves more than in our example (processes exchange messages), but where rule (NEW) is always executed before rule (SUCCESS) can take place.

10.2 Termination and the Boomerang Game

Since we cannot be sure that the Paxos algorithm terminates, in this section we propose a strategy that, under some specific conditions, ensures Termination of Paxos with probability 1.

In distributed algorithms, a common way of overcoming the problem of starting a new round too early with respect to the speed of message transmission and process computation is to progressively increase the duration of a round, i.e., the interval of time between the beginning of a round and the beginning of the following one. The idea behind this strategy is that, if we suppose that the network is working (more or less) correctly and that the other processes are not all crashed, then the leader, by waiting longer, increments its probability of being able to receive answers from other processes. This strategy is used for $\diamond S$ failure detector algorithms [12] and we can imagine using it also for Paxos.

In order to stay general and free ourselves from algorithmic details, we represent the problem as a game, and we give a condition that allows us to affirm that the game ends with a victory with probability 1.

The Game An Australian Aborigine wizard is playing boomerang. With his magical powers he can create an infinity of boomerangs. It can be that some boomerangs are lost after being launched, but all boomerangs that are launched after a certain point in time (let us call it t and suppose it is finite) eventually come back. The wizard knows about this property, but he can never tell whether time t has passed or not. Moreover, for boomerangs that come back, it is impossible to know in advance how long it takes for them to return. The game proceeds by rounds. A round r of the game consists in launching a boomerang tagged with r and waiting for its return. If the boomerang does not return after a certain period of time, the wizard assumes it has been lost and launches a new boomerang, thus starting a new round, this time waiting one minute longer than in the previous round, etc. Any boomerang that was launched in a round smaller than r and that arrives after the wizard has launched round r boomerang is immediately discarded. The wizard wins (and the game stops) when he receives the exact last boomerang he has sent.

We can prove that the wizard is almost certain to win the game if the return probability distribution is the same for all boomerangs. In other words, if we assume that the probability that boomerang r' comes back s seconds after being launched is the same as the probability that boomerang r'' comes back s seconds after being launched, we can be sure that the game terminates with probability 1.

The analogy of the game with the Paxos algorithm and $\Diamond S$ failure detector algorithms is the following.

Paxos We can see the wizard as the eventual only leader of the algorithm and the boomerangs as the messages that the leader exchanges with the other processes in the system. The fact that boomerangs can get lost represents the fact that communication channels are lossy and that receivers can crash. Time t indicates the combination of two things: that the network eventually works properly (it does not lose messages any longer and it ensures their eventual delivery) and that there is a majority of correct processes (a majority of answer messages is sent). The fact that the wizard cannot know after how long the boomerangs come back represents the fact that the system is asynchronous and the leader cannot forecast neither how long it takes for its message to reach the destination process and for the answer message to come back to him, nor how long it takes for other processes to deal with the message. The wizard winning the game corresponds to the fact that the leader receives answers from a majority of processes for the last request it has sent.

Failure Detectors In this case, we can see the wizard as a $\Diamond S$ failure detector and the boomerangs as the messages that the failure detector exchanges with the other failure detectors in the system. In this case, since $\Diamond S$ failure detectors are usually employed with non-lossy channels, boomerangs never get lost (time t coincides with 0). The fact that the wizard cannot know after how long the boomerangs come back represents the fact that the system is asynchronous and the failure detector cannot forecast neither how long it takes for its message to reach the destination process and for the answer message to come back to it, nor how long it takes for other failure detectors to deal with the message. The wizard winning the game corresponds to the fact that the failure detector receives answers from all the failure detectors of the processes that are not crashed at that time.

10.2.1 Formalization

We consider a discrete time, informally counted in minutes, represented by the set $\mathbb{N}$ of natural numbers, and we consider also rounds as belonging to the set $\mathbb{N}$ of natural numbers. We define the following probabilities.

- $\text{P}_{\text{AT}}(r, i)$: the probability that the boomerang launched at round r comes back exactly *at* the i-th minute after being launched,
- $\text{P}_{\text{BEFORE}}(r, k)$: the probability that the boomerang launched at round r comes back *before* k minutes have passed from its launch,
- $\text{P}_{\text{FAIL}}(r)$: the probability that the boomerang launched at round r does not come back before the wizard launches the next one,
- $\text{P}_{\text{CONTINUE}}(r)$: the probability that the game does not end at round r,
- $\text{P}_{\text{NEVERENDING}}$: the probability that the game does never end.

Since we are going to examine an infinite time, and since we know that there is a (finite) time t after which no boomerang gets lost any longer, then we can (without loss of generality) translate the time axis and place its origin at the first time greater than or equal to t when a new boomerang is launched. In this way, we can assume that every boomerang that is launched during our observation always returns, which translates into the following formula:

$$\text{for all } r \in \mathbb{N}, \sum_{i=0}^{\infty} \text{P}_{\text{AT}}(r, i) = 1 \tag{10.1}$$

The probability that a boomerang returns *before* k minutes from the launching time have passed can be computed starting from the probability of return *at* a certain time (after launching time): the boomerang comes back before k if it comes back at some time i comprised between 0 and k.

$$\text{P}_{\text{BEFORE}}(r, k) = \sum_{i=0}^{k} \text{P}_{\text{AT}}(r, i) \tag{10.2}$$

If we assume that the wizard waits for the boomerang during wait(r) minutes at round r, the probability of not succeeding at round r is:

$$\text{P}_{\text{FAIL}}(r) = 1 - \text{P}_{\text{BEFORE}}(r, \text{wait}(r)) \tag{10.3}$$

By definition, the game continues at round r if none of the first r rounds was successful (if all first r rounds failed). Since, once a new boomerang is launched, the coming back of any of the previous boomerangs is completely irrelevant, we can assume that the event of failing at a certain round is completely uncorrelated with the event of failing at any other round. This allows us to express the probability of having to continue playing after round r as a simple product of probabilities of having to continue playing after all rounds smaller than or equal to r.

$$\mathrm{P}_{\mathrm{CONTINUE}}(r) = \prod_{k=0}^{r} \mathrm{P}_{\mathrm{FAIL}}(k) = \prod_{k=0}^{r} (1 - \mathrm{P}_{\mathrm{BEFORE}}(k, \mathsf{wait}(k))) \tag{10.4}$$

Finally, the game does not end if it continues forever.

$$\begin{array}{lcl} \mathrm{P}_{\mathrm{NEVERENDING}} & = & \lim_{r\to\infty} \mathrm{P}_{\mathrm{CONTINUE}}(r) \\ & = & \lim_{r\to\infty} \prod_{k=0}^{r} \mathrm{P}_{\mathrm{FAIL}}(k) \\ & = & \lim_{r\to\infty} \prod_{k=0}^{r} (1 - \mathrm{P}_{\mathrm{BEFORE}}(k, \mathsf{wait}(k))) \\ & = & \prod_{k=0}^{\infty} (1 - \sum_{i=0}^{\mathsf{wait}(k)} \mathrm{P}_{\mathrm{AT}}(k, i)) \end{array}$$

10.2.2 Proofs

If the return probability distribution is the same for all boomerangs, the strategy that consists in waiting at each round for one minute longer than in the previous round ensures the termination of the game almost always. In other words, the probability that the game never ends is zero.

We start by recalling a couple of theorems on series of numbers.

Theorem 10.2.2.1 (Sub-series) *Let $(x_i)_{i\in\mathbb{N}}$ and $(y_j)_{j\in\mathbb{N}}$ be two series.*

$$\lim_{i\to\infty} x_i = l \textit{ and } \lim_{j\to\infty} y_j = \infty \textit{ implies } \lim_{j\to\infty} x_{y_j} = l$$

Proof. We prove that for all $\epsilon > 0$, there exists n such that for all $j > n$, $|x_{y_j} - l| < \epsilon$.

Let $\epsilon > 0$. $\lim_{i\to\infty} x_i = l$ implies there exists m such that for all $i > m$, $|x_i - l| < \epsilon$. $\lim_{j\to\infty} y_j = \infty$ implies there exists n such that for all $j > n$, $y_j > m$. $y_j > m$ implies $|x_{y_j} - l| < \epsilon$. □

Theorem 10.2.2.2 (Partial products) *Let $(x_k)_{k\in\mathbb{N}}$ be a series.*

$$\lim_{k\to\infty} x_k = 0 \textit{ implies } \lim_{r\to\infty} \prod_{k=0}^{r} x_k = 0$$

Proof. The notation $\lim_{k\to\infty} x_k = 0$ is an abbreviation for the following statement.

$$\forall\, \epsilon > 0,\ \exists\, n \in \mathbb{N},\ \forall\, k > n,\ x_k < \epsilon$$

Let $\epsilon < 1$ and n such that for all $k > n$, $x_k < \epsilon$. Let $r > n$ and pose $C \triangleq \prod_{k=0}^{n} x_k$. We have:

$$\prod_{k=0}^{r} x_k = \prod_{k=0}^{n} x_k \times \prod_{k=n+1}^{r} x_k = C \times \prod_{k=n+1}^{r} x_k$$

Since $x_k < \epsilon$ for $k > n$, we have:

$$\prod_{k=0}^{r} x_k < C \times \epsilon^{(r-n)}$$

Since $\epsilon < 1$,

$$\lim_{r\to\infty} C \times \epsilon^{(r-n)} = \lim_{r\to\infty} C \times \epsilon^{r} = C \times \lim_{r\to\infty} \epsilon^{r} = C \times 0 = 0$$

The series $(\prod_{k=0}^{r} x_k)_r$ is bounded (after n) by a series converging to 0. By consequence,

$$\lim_{r\to\infty} \prod_{k=0}^{r} x_k = 0$$

□

Now we have all the elements for proving that, if boomerangs all have the same return probability distribution, then there exists a winning strategy for the wizard.

Theorem 10.2.2.3 *Assume that, for all $r \in \mathbb{N}$ and for all $i \in \mathbb{N}$, $P_{\text{AT}}(r,i) = P_{\text{AT}}(0,i)$ and $\lim_{r\to\infty}$ wait$(r) = \infty$. Then, $P_{\text{NEVERENDING}} = 0$.*

Proof. From Equation 10.1, we know that for all $r \in \mathbb{N}$ $\sum_{i=0}^{\infty} \mathrm{P}_{\text{AT}}(r,i) = 1$. This is valid also for $r = 0$, which gives us $\sum_{i=0}^{\infty} \mathrm{P}_{\text{AT}}(0,i) = 1$. By definition of infinite sums, we can pass to the limit $\lim_{k\to\infty} \sum_{i=0}^{k} \mathrm{P}_{\text{AT}}(0,i) = 1$. By Theorem 10.2.2.1 with hypothesis $\lim_{r\to\infty} \text{wait}(r) = \infty$ we have that

$$\lim_{k\to\infty} \sum_{i=0}^{\text{wait}(k)} \mathrm{P}_{\text{AT}}(0,i) = 1.$$

Since $\mathrm{P}_{\text{AT}}(k,i) = \mathrm{P}_{\text{AT}}(0,i)$, we can write $\lim_{k\to\infty} \sum_{i=0}^{\text{wait}(k)} \mathrm{P}_{\text{AT}}(k,i) = 1$. By Equation 10.2, we know that $\mathrm{P}_{\text{BEFORE}}(k, \text{wait}(k)) = \sum_{i=0}^{\text{wait}(k)} \mathrm{P}_{\text{AT}}(k,i)$, and thus

$$\lim_{k\to\infty} \mathrm{P}_{\text{BEFORE}}(k, \text{wait}(k)) = 1.$$

From this we trivially get that $\lim_{k\to\infty}(1-\mathrm{P}_{\text{BEFORE}}(k, \text{wait}(k))) = 0$. And applying Equation 10.3 we obtain $\lim_{k\to\infty} \mathrm{P}_{\text{FAIL}}(k) = 0$. By Theorem 10.2.2.2, we get

$$\lim_{r\to\infty} \prod_{k=0}^{r} \mathrm{P}_{\text{FAIL}}(k) = 0.$$

Which by Equation 10.4 becomes $\lim_{r\to\infty} \mathrm{P}_{\text{CONTINUE}}(r) = 0$. Finally, since $\mathrm{P}_{\text{NEVERENDING}} = \lim_{r\to\infty} \mathrm{P}_{\text{CONTINUE}}(r)$, we obtain $\mathrm{P}_{\text{NEVERENDING}} = 0$. $\square$

On the other hand, if boomerangs can have arbitrary probabilities of coming back (even though still respecting the condition $\sum_{i=0}^{\infty} \mathrm{P}_{\text{AT}}(r,i) = 1$ for all r), then there is no general waiting strategy capable of ensuring that the game ends with probability 1.

Theorem 10.2.2.4 *There exists no strategy wait(r) such that for any probabilities $P_{\text{AT}}(r,i)$ we have $P_{\text{NEVERENDING}} = 0$.*

Proof. Suppose there exists such a strategy. It is easy to construct a probability $\mathrm{P}_{\text{AT}}(r,i)$ such that $\mathrm{P}_{\text{NEVERENDING}} \neq 0$. Just take the distribution that is null everywhere except in point wait$(r) + 1$ when it equals 1. The probability of success at each round is null and consequently the probability of an infinite game is 1. $\square$

Chapter 11

Chandra-Toueg Versus Paxos

From the ways they are presented in the respective papers, and from the settings in which they work, the Chandra-Toueg and the Paxos algorithms look very different from each other. However, using our formalism it is pretty easy to remark a big number of similarities. In §7.1.2 we have already remarked that the Paxos answer to a *prepare request* is the same as the CT message P1, since both of them are sent to the round leader [1] and contain the current belief of the sending process, the Paxos *accept request* is the same as the CT message P2, since both of them are sent by the round leader and contain the best among the beliefs received by the leader, the Paxos answer to an *accept request* is the same as an acknowledging CT message P3, since both of them are sent to the their round leader and contain a positive confirmation of message reception.

Another similarity is given by some of the phases through which every leader has to go: phase P2 of CT resembles phase waiting of Paxos, since in both of them the leader is waiting to receive enough (a majority of) P1 messages; phase P4 of CT resembles phase polling of Paxos, since in both of them the leader is waiting to receive enough (a majority of) P3 messages.

Of course, the algorithms are not completely identical. For example, a difference is given by the behavior of the participants. In fact, while in CT every process proceeds sequentially (the coordinator goes through phases P1, P2, P3, P4, W and simple participants go through phases P1, P3, W), in the Paxos algorithm simple participants just wait for messages from leaders and only leaders make progress in a sequential way (going through phases trying, waiting, polling, done). Paxos trying does not have any correspondence in CT because coordinators do not need to send P0

[1] We use here the name leader to indicate both the Paxos leader and the CT coordinator.

messages. done, as we explained in §7.1 was artificially inserted in order to mark the distinction between a leader that has gone until the end of the round and a leader that is simply idle.

In what follows, we take the rules of Tables 6.2 and 7.2, we put them side by side and we mark down similarities and differences.

11.1 Rule Comparison

If we consider the number of rules that appear in Tables 6.2 and 7.2, we count eight rules for CT and seven for Paxos. However, if we look closer, we see that CT has two pairs of rules, (PHS-3-TRUST) and (PHS-3-SUSPECT), (PHS-4-FAIL) and (PHS-4-SUCCESS), that are related to the possible eventualities of phases P3 and P4. Thus, if we agree that each pair counts as one single rule, CT has six rules, while Paxos has seven. If, as we said, CT and Paxos are very similar to each other, why do we find such a discrepancy in the number of rules? The answer is quite simple, and we will give it while comparing the rules.

The first rules we encounter are (WHILE) and (NEW). Both of them can be executed only by processes that have not yet decided and both of them increase the round number, by one in the case of CT and to the next request number usable by the process in Paxos. In CT, any process in phase W can execute (WHILE) whereas in Paxos only leaders can execute (NEW), regardless of their current phase. These differences are due to the fact that in CT any process (and not just the coordinators) advances in subsequent phases and, once all the phases of a round have been visited, the process has to go to the following round. In Paxos instead only the leader has the right to start a new round. Since the setting in which Paxos is designed to work do not give any guarantee on message delivery and non-crashed processes, the algorithm could deadlock at any time if the leader waited forever for messages that it would never get the chance to receive. Thus, in order to guarantee the fault tolerance of Paxos, the leader has to be able to start a new round at any time.

Paxos rule (SND-PREP-REQ) can be executed by any leader in phase trying. It sends a message to every process in the system, asking for beliefs, and changes the leader's phase to waiting. The target of this rule is to inform all the processes in the system that a new round has started and that the current leader is waiting for beliefs. This rule does not have a counterpart in CT because, in the Chandra-Toueg algorithm, processes proceed sequentially in rounds, can identify at any moment in time the coordinator of any round and spontaneously send it their belief without being asked to do so.

With both rule (PHS-1) and (RCV-PREP-REQ) the process sends its current belief to the leader of the round. As we said, processes in CT execute (PHS-1) spontaneously right after starting a new round, while Paxos processes execute (RCV-PREP-REQ) upon receipt of an explicit request from a leader.

With both rule (PHS-2) and (SND-ACC-REQ) the round leader selects the best proposal among the majority of beliefs it has received and communicates such value by sending it to all processes in the system.

Executing rule (PHS-3-TRUST) and (RCV-ACC-REQ) the process can acknowledge the reception of the best proposal for the round by replying to the leader either with an ack (CT) or with the value itself (Paxos). Rule (PHS-3-SUSPECT) of CT is used to inform the coordinator that the process is moving forward since the message containing the best proposal has not been received. This rule is necessary for the liveness of CT because, as we have already said, processes proceed sequentially and can block in some phases if they do not receive the messages they are waiting for or they are not left free to advance to the next phase if they fear the coordinator has crashed. Rule (PHS-3-SUSPECT) has no counterpart in Paxos because processes never block waiting for messages (rule (NEW) is always enabled so there is always a way for the processes to advance), thus, when a participant answers a request in Paxos, it always answers in a positive way.

A similar reasoning applies for (PHS-4-FAIL). This rule is needed in the Chandra-Toueg algorithm in order to take into account the possibility that the coordinator receives negative answers to its P2 message. Since in Paxos, if there are answers, they are positive, rule (SUCCESS) is the counterpart of (PHS-4-SUCCESS): if the leader receives a majority of acknowledgments, it broadcasts the decision value.

The (RBC-DELIVER) rule is exactly the same both for CT and for Paxos.

11.2 Structural Differences

Looking at configurations, we see that both CT and Paxos have three components, and that there is a correspondence between them: the first component represents the Reliable Broadcast abstraction, the second component represents the network abstraction and the third component represents the process states. The Reliable Broadcast component is the same in both formalizations since it is independent from network losses/replications and from process crashes. The network abstractions are similar, but the Paxos one is slightly more complex than the one in CT. We can remark the difference already from the names that we have given to the configuration second component: a general **C** (from Channel) for Paxos and a more specific **Q** (from Quasi-reliable point to point) for CT. In fact, even though the message structure is the same in both channels (round, phase, sender, receiver, content), a quasi reliable point to point channel has only three transmission states (outgoing, transit, received) while a general channel can also replicate and lose transiting messages. So it is necessary to adapt the tag shape in order to be able to represent these two additional transmission states. Moreover, quasi reliable channels have specific properties in a run (every message sent from a correct process to a correct process must be received) that are absolutely unknown to general channels. Process states are quite similar, but once again Paxos is more complex because for each process it has to say explicitly whether the process is leader or not (in CT this is inferred from the round number). Moreover, since processes can crash and recover, and since at recovery they must be able to remember some important data, to represent a crash we cannot simply say that the whole process state equals $\perp$, as we do with CT, but we need to keep all the information and use a special field inside the state component for indicating the crash. Also, for each process in Paxos, a new (with respect to CT) entry has to be introduced to keep track of the incarnation number.

11.3 Proof Comparison

The proofs of correctness of both the Chandra-Toueg and the Paxos algorithms proceed exactly following the same guideline. For both of them, we first prove a number of properties stating that it is possible to establish an order among program counters, among states and among message histories. Moreover, we prove that the process beliefs are always non-decreasing in time and smaller than (equal to) the current round number. We also show that the proposal of a process in a round always equals the last value the process has acknowledged in a previous round and carries as stamp that round number. In this preliminary properties, there is only a small difference between the two algorithms. Since CT proceeds sequentially, for this algorithm we can prove that, if we take the state of a process in a certain point of a run, then we are sure that, in that particular run, the process has passed through all previous valid program counters it traversed. For Paxos, instead, we cannot state this kind of properties, as the algorithm is not strictly sequential. However, this does not pose any problem because the reason why we need to go back in history in CT is to find positively acknowledging messages, which can be followed by many negative ones. In Paxos, instead, we know that only ”positive” messages are sent (if a participant refuses the leader proposal, it simply does not answer), so the last acknowledging message is simply the last message sent.

After providing all the elementary properties, we come to the Validity and Agreement proofs. Validity is proved in exactly the same way for both cases, with the only difference of the name of the rule under exam: (PHS-2) for CT and its corresponding Paxos version (SND-ACC-REQ). For Agreement, the structure of the proofs is the same, with the definition of locked value and the proof that two locked values must necessarily be the same. A small portion of the Paxos proof results slightly more complex than the corresponding CT one because of the lack of sequentiality of Paxos that forces us to split into two separate cases the problem of knowing ”what is the next round in which a process takes part”.

A completely different reasoning is instead done for Termination. In fact, while the Chandra-Toueg algorithm, with its sequentiality, ensures that Termination is reached after a certain number of properties of the underlying modules become true, the Paxos algorithm proves to have non-terminating runs even if the system presents all the best conditions we can imagine. This last aspect opens a tricky question.

11.4 Is Paxos a Consensus Algorithm?

Although nobody has explicitly given a definition in literature (or, at least, we could not find any), it is common and accepted knowledge that the Consensus problem requires the guarantee of three properties: Validity, Agreement and Termination. Besides, the most fundamental result on distributed Consensus, the FLP impossibility result, states that Consensus cannot be reached in an asynchronous system where even a single process can crash. The proof of this theorem consists in showing that, even in presence of a single crash, no algorithm can guarantee to contemporaneously respect the properties of Validity, Agreement **and** Termination. In particular, if one wants to make sure to respect Agreement, then no guarantee can be given on the respect of Termination. This can be intuitively explained by saying that, until we do not have the certainty that a non-responding process i has crashed, we cannot take the responsibility of deciding for a value since, if i has not crashed, we could end up deciding for two different values. Thus, we have to wait for the certainty that the non-responding process has crashed. But since the system is asynchronous, we cannot be sure that this certainty will ever come. And we might end up waiting forever. As we have seen with the Chandra-Toueg algorithm, a common way of bypassing the FLP impossibility result is to add some synchrony to the system, typically with the use of failure detectors.

The problem with Paxos gets more complicated because the algorithm is designed to work with completely unreliable channels and with crash-recovery model failures. Even though the fact that crashed processes can come back to life might look as an advantage, it actually is not, since a process that crashes ”forgets” about all the previously received messages and thus, when it recovers, it cannot know what was the last action it was supposed to do and resume the execution from that point. Moreover, the fact that crashed processes can recover eliminates the requirement, fundamental for CT, that at any moment of a run there is a majority of correct processes.

From here, and from the fact that Paxos is designed to work with completely unreliable communication springs the necessity of making the algorithm non-blocking and somehow independent on the messages exchanged between the processes. Of course, since from a safety point of view it is fundamental to ensure the properties of Agreement, the constraints of gathering a majority of beliefs before making a proposal and of not taking any decision until the majority of processes has agreed on a value have both been maintained. But, as we have seen, there is no guarantee of being able to gather these majorities of messages. Therefore, the beginning of a new round cannot be put under the condition of receiving a majority of messages from other processes (like in CT) because the algorithm would run the risk of deadlocking. Thus, the choice made in Paxos is to let it start new rounds in a completely asynchronous and unconditioned manner. It is precisely this unconditioned rule, which represents the biggest difference between CT and Paxos, that allows us to provide non-terminating runs of the algorithm.

Our concern about Termination is not a mere theoretical thought, but it is also related to practical everyday life. In fact, there are many real implementations of Paxos, for all sorts of problems, running at this moment in the world. Depending on the implementer, the restarting condition can vary from being completely random to specifying some enabling event (for example a timeout). However, in a completely asynchronous system, given any restarting condition we are able to provide a run that does not terminate. Thus, the implementations of Paxos that are placed into completely asynchronous settings seriously run the risk of not terminating, and the fact that it has not happened yet is not a guarantee for the future.

At this point, the question becomes: is it possible to give a condition that prevents Paxos from restarting arbitrarily and that can lead the algorithm to Termination?

We have already given a tentative answer to this question in §10.2. If the additional condition forces the leader to always wait for a time interval that grows bigger and bigger with the growth in number of unsuccessful rounds, and if the probability distribution of receiving answers from other processes stays constant, then we can prove Termination with probability 1. However, as we have shown, this "longer waiting" condition does not work for all general cases of probability distribution. Besides, we are not yet sure we can assume that a system has a constant probability distribution, since it seems more intuitive to suppose the contrary. Moreover, with this condition, we cannot prove that Termination is guaranteed, but only that non-terminating runs happen with a frequency that is negligible with respect to the number of terminating ones.

By analyzing the algorithm, we see that the most intuitive constraints to impose on the system in order to achieve Termination would be to eventually have

- at least a majority of processes that are alive in the same time interval,
- a quasi-reliable network,
- the knowledge of the messages' round trip transmission time.

The quasi-reliable network would ensure that all messages sent between correct processes are eventually received. The knowledge of the round trip time would allow the leader to start a new round only after this time has passed, thus giving the possibility to all existing round messages to be received. Finally, the presence of at least a majority of correct processes ensures that the quorum of messages needed for establishing a proposal and for broadcasting a decision is reached.

However, we think that these settings are far too demanding for an algorithm that was presented as designed to work with lossy channels and crash-recovery model failure. In fact, the first two items are the same required by the Chandra-Toueg algorithm and they are quite far from the unconstrained typical Paxos settings we just mentioned. The third item above, is even a stronger requirement because it basically requires the system to eventually become synchronous.

Thus, going back to the question that constitutes the title of this section, we think that the Paxos algorithm cannot be considered a Consensus algorithm because it is not capable of guaranteeing the Termination property that, as far as we have seen, is a necessary condition to solve the Consensus problem.

With this, we are not implying that it is a priori impossible that the Paxos algorithm will ever ensure Termination. What we want to say is that the weakest system settings under which Paxos is guaranteed to terminate have still to be fixed, and when this will be done Paxos should be presented as an algorithm capable of solving Consensus in *those* settings and not any longer in the ones in which it is presented today.

Chapter 12

Related Work

Given the amplitude of the domain, the aim of this chapter is not to present an exhaustive description of all existing work on verification of distributed algorithms, but rather to provide samples of particularly relevant, recurrent or well know formalisms, methods and structures.

12.1 Our Related Work

We present here some of the introductory work that we have done in the domain of distributed Consensus formalization. The first is a formalization of Failure Detectors that is alternative to the one presented in §5.4. The other is a first attempt of formalization through the use of a process calculus.

12.1.1 Formalization of Failure Detectors

Together with Uwe Nestmann, we proposed a formal definition of Failure Detectors via operational semantics [43]. We started from the consideration that a distributed system is composed of an environment Γ and a group N of processes. Then, any action of the system falls under one of the three following categories.

(ENV): a modification of the environment;
(TAU): a step taken by any non-crashed process;
(SUSPECT): the suspicion by a non-crashed process against another process.

The abstract rules illustrating these three categories are shown in Table 12.1.

$$(\textsc{env})\ \frac{\text{"failure detectors events happen in the environment"}}{\Gamma\ \rightarrow \Gamma'}$$

$$(\textsc{tau})\ \frac{\Gamma\ \rightarrow \Gamma' \qquad N \xrightarrow{\tau@i} N' \qquad \text{"}i\text{ not crashed in }\Gamma\text{"}}{\Gamma \vdash N\ \rightarrow \Gamma' \vdash N'}$$

$$(\textsc{suspect})\ \frac{\begin{array}{c}\Gamma\ \rightarrow \Gamma' \qquad N \xrightarrow{\text{susp}_j@i} N' \\ \text{"}i\text{ not crashed in }\Gamma\text{"} \qquad \text{"}j\text{ may be suspected by }i\text{ in }\Gamma\text{"}\end{array}}{\Gamma \vdash N\ \rightarrow \Gamma' \vdash N'}$$

Table 12.1: Operational Semantics Scheme

Failure Detectors enter the description in rule (SUSPECT). In fact, they limit the possible executions of the rule by imposing conditions related to the environment. This is what is called "j may be suspected by i in Γ" in rule (SUSPECT) of Table 12.1.

We then suggested to represent the environment with a pair: the first component, **TI**, records the *trusted-immortal* processes, while the second component, **C**, records the crashed processes. Whenever a process becomes *trusted-immortal* it means that, after that point in time, the process will not crash and no other process will ever suspect it. Whenever a process enters the set **C** it means that the process has crashed and will stay so forever [1]. Thus, we can express the rules (ENV) and (TAU) as in Table 12.2, and we can model all the categories of Failure Detectors as in Table 12.3. Note that, for any single (ENV) step, there can be many processes becoming trusted-immortals (the ones in set TI) or crashed (the ones in set C) at the same time.

This description is the counter part of what we presented in §5.4, as it better describes from an operative point of view what happens in the system in "real time". Our previous description, instead, approaches the problem from a static point of view: it supposes that we know all the entire possible runs and it simply dives inside them to find (if it exists) the point in time after which the chosen failure detector properties are satisfied.

[1]We consider only crash model failures.

$$(\textsc{env})\ \frac{(\mathbf{TI} \cup \mathrm{TI}) \cap \mathrm{C} = \emptyset \qquad (\mathbf{C} \cup \mathrm{C}) \cap \mathrm{TI} = \emptyset \qquad |\mathbf{C} \cup \mathrm{C}| \leq \mathrm{maxfail}}{(\mathbf{TI}, \mathbf{C}) \ \rightarrow (\mathbf{TI} \uplus \mathrm{TI}, \mathbf{C} \uplus \mathrm{C})}$$

$$(\textsc{tau})\ \frac{(\mathbf{TI}, \mathbf{C}) = \Gamma \ \rightarrow \Gamma' \qquad N \xrightarrow{\tau @ i} N' \qquad i \notin \mathbf{C}}{\Gamma \vdash N \ \rightarrow \Gamma' \vdash N'}$$

Table 12.2: Rules (ENV) and (TAU) expressed with Reliable Information

$$(\mathcal{P}/\mathcal{Q}\text{-}\textsc{suspect})\ \frac{(\mathbf{TI}, \mathbf{C}) = \Gamma \ \rightarrow \Gamma' \qquad N \xrightarrow{\mathrm{susp}_j @ i} N' \qquad i \notin \mathbf{C} \qquad j \in \mathbf{C}}{\Gamma \vdash N \ \rightarrow \Gamma' \vdash N'}$$

$$(\mathcal{S}/\mathcal{W}\text{-}\textsc{suspect})\ \frac{(\mathbf{TI}, \mathbf{C}) = \Gamma \ \rightarrow \Gamma' \qquad N \xrightarrow{\mathrm{susp}_j @ i} N' \qquad i \notin \mathbf{C} \qquad j \notin \mathbf{TI} \neq \emptyset}{\Gamma \vdash N \ \rightarrow \Gamma' \vdash N'}$$

$$(\Diamond\text{-}\textsc{suspect})\ \frac{(\mathbf{TI}, \mathbf{C}) = \Gamma \ \rightarrow \Gamma' \qquad N \xrightarrow{\mathrm{susp}_j @ i} N' \qquad i \notin \mathbf{C} \qquad j \notin \mathbf{TI}}{\Gamma \vdash N \ \rightarrow \Gamma' \vdash N'}$$

Table 12.3: Formalization of Different Classes of Failure Detectors

12.1.2 An Attempt Using Process Calculi

Some years ago, together with Uwe Nestmann and Massimo Merro, we did a first attempt of formalization of the Chandra-Toueg Consensus algorithm through the use of process calculi [44]. We chose a distributed asynchronous value-passing process calculus that we equipped with an operational control over crash-failures and Failure Detector properties. We developed a global-view matrix-like representation of reachable states that contained the complete history of messages sent ”up to a certain point of the execution”. We provided a formal semantics for such reachable states abstraction and we used it instead of the local-view semantics to prove the Consensus properties. This kind of approach was substantiated by the presence of a tight operational correspondence proof between the local-view process semantics and the global-view matrix semantics. However, the gap between the two semantics was so big that the operational correspondence proof resulted more complex than the correctness proofs of the algorithm (that were instead our original target). This is the reason why we preferred to explore another direction rather than continuing with process calculi.

12.2 Other Formalisms

The idea of using formal methods for describing and proving correctness of distributed algorithms is not new. We present here some of the main theories that have been developed in this field since the end of the Eighties.

12.2.1 I/O Automata

In 1987, the feeling that the description and proof of distributed algorithms needed to be done in a more rigorous way pushed Lynch and Tuttle to introduce I/O Automata [40, 55]. The Input/Output (or I/O) Automata model is based on the composition of (possibly infinite-state) nondeterministic automata where transitions are labeled with the names of the process actions they represent and executions are sequences of alternating states and actions. The actions are partitioned into sets of input and output actions, as well as internal ones representing internal process actions. To give the model the event-driven behavior that is characteristic of asynchronous distributed algorithms, actions are split into actions locally-controlled by the system (output and internal) and actions controlled by the system’s external environment (input). Since the system must be able to accept any input at any time, the input actions have the unique property of being enabled from every state. This can give origin to a flood of inputs that could prevent the automata from taking any output or internal actions. Lynch and Tuttle introduce then the concept of *fairness*: a computation of a system is said to be fair if every system component has always eventually the chance to take a step. Given the communication based characteristic of I/O Automata with respect to their environment, a problem to be solved corresponds to a sequence of input and output actions considered acceptable. And a solution to a problem is an automaton that exhibits only behaviors that are fair and that are contained in the set of acceptable behaviors.

Proofs of correctness of an algorithm are typically done by expressing both the algorithm and the desired behavior with I/O Automata, and then showing that the automaton representing the algorithm *solves* the automaton representing the specification. An automaton A *solves* an automaton B if every correctness condition satisfied by the fair behavior of B is satisfied by the fair behavior of A and the relation between executions of A and executions of B is done through a mapping, the *possibilities mapping*, that is reminiscent of bisimulation from CCS [41].

Many extended or specialized variants of I/O Automata have then appeared, for example Hybrid I/O Automata [39] for the verification of systems that contain a combination of continuous and discrete behavior, or Timed I/O Automata [33] for the verification of real-time systems, or Probabilistic I/O Automata [50, 51, 57] for the verification of certain randomized distributed algorithms.

The I/O Automata approach is quite far from ours. However, sometimes I/O Automata proofs use the same strategy as we do: stating and proving sets of invariants for the runs and using them to deduce the global properties of the algorithms.

12.2.2 UNITY

In their book of 1988, Chandy and Misra proposed a theory - a computational model and a proof system - called Unbounded Nondeterministic Iterative Transformations (UNITY) [13]. The theory attempts to decouple the programmer's thinking about a program and its implementation on an architecture, i.e. to separate the concerns of *what* (should be done) from those of *when*, *where* or *how*.

A UNITY program consists of a declaration of variables, a specification of their initial values, and a set of statements. Assertions over the statements of a program are of two kinds: universal and existential. This is enough to reason about the two kinds of program properties: safety and progress. A program execution starts from any state satisfying the initial condition and goes on forever; in each step of execution some assignment stated is selected nondeterministically and executed. Nondeterministic selection is constrained by the ”fairness” rule: every statement is selected infinitely often, and fairness is captured using existential quantification. Since an execution of a UNITY program is an infinite sequence of statement executions, ”terminating” programs are represented through the concept of fixed-point. A fixed-point of a program, if it exists, is a program state such that execution of any statement in that state leaves the state unchanged.

The proof of correctness of an algorithm is divided into two parts. First one lists and proves all the invariants of the program, i.e. the predicates that are true at all points during program execution. Then one uses a combination of the invariants and other properties to prove that the claim is correct.

Even though we did not know UNITY at the time we designed our formalization, we can find a number of common points between them. First of all, both approaches declare a set of variables, a specification of their initial values and a set of statements (in our case the inference rules). Also, in both cases the execution proceeds by steps, where the execution of a statement can change the value of some variables. Moreover, both approaches prove correctness of the algorithms by initially stating and proving a set of invariants and then using a combination of such invariants to prove the desired properties. A curious thing to notice is that a terminating UNITY program is defined as an infinite ”run” for which the execution of any statement leaves the state unchanged. Our CT Termination proof, instead, shows that terminating runs are finite. However, this discrepancy is present in the specific case of CT, due to the structure of the particular algorithm rules, but it is not present in Paxos. In fact, if we suppose that we can reach Termination in Paxos (i.e., all alive processes eventually decide) there remains still a number of rules, independent of the decision field of processes, that can be executed forever without affecting the capital part of the state, i.e., the decision field. Thus a similarity between UNITY and our formalization exists also at the level of definition of termination.

12.2.3 TLA+

TLA+ [37] is a language for specifying and reasoning about concurrent and reactive systems. It allows to write formal descriptions of distributed systems, particularly asynchronous ones. It is issued from the Temporal Logic of Actions (TLA), which in turn was issued from Pnueli's Temporal Logic. TLA+ specifies a system by describing its allowed behaviors through a formula and provides a way to formalize the style of reasoning about systems known as assertional reasoning.

The execution of a system is represented as a sequence of states. A state is an assignment of values to variables. Specifying a system in TLA+ means specifying a set of possible behaviors, i.e. the ones representing a correct execution of the system. TLA+ specifications are partitioned into modules, where each module describes a particular component of the system or defines an operator used in other modules. Proofs are done using proof rules, deducing true formulas from other true formulas.

This approach is quite different from ours in that it is based on temporal logics and all the reasoning is done on logic formulas.

12.2.4 Process Calculi

Since their introduction in the Seventies, process calculi have "colonized" many domains and proved their efficiency in describing many kinds of problems. Some attempts have been done also in the domain of distributed algorithms.

The main difference with our approach is that our description is syntax-free, while all process calculi need, by definition, to specify both the syntax and the semantics of the language.

μ-CRL

μ-CRL [26, 27] is a process algebraic language that is designed to study description and analysis techniques for (large) distributed systems. Starting from the criticism that basic process algebras such as CCS [41] and CSP [29] lack the ability of handling data when studying behavioral properties of many realistic systems, μ-CRL is especially developed to take account of data in the study of communicating processes. The language μ-CRL consists then of data and processes. Data are defined by means of equational abstract data types. The user can declare types and functions that work on such types, and describe the meaning of these functions by equational axioms. Processes are described in the style of Algebra of Communicating Processes (ACP) [7]. They consist of a set of uninterpreted actions that may be parametrised with data and composed with sequential, alternative and parallel composition operators. Moreover a conditional (if-then-else construct) can be used to have data influence the course of a process, and alternative quantification is added to sum over possibly infinitely many data elements of some data type.

An implementation of a communication protocol can be described as the parallel composition of several components $C_1 \cdots C_n$. Components communicate via internal actions, resulting in internal communications. The specification that the implementation should satisfy is given by a process *Spec*. In the μ-CRL process algebraic framework, satisfying a specification means being equal to it, therefore proving correctness of a communication protocol means showing that the parallel of the components (under some restrictions and conditions) equals the specification. The strategy is a refined version of the strategy that consists on taking an equation G and showing that both the parallel composition of the components and the *Spec* are solutions of G. The refinements of this strategy are based on linear process operators, a particular format for the notation of processes. This format is similar to the UNITY format of [13] and to the precondition/-effect notation of [40], but it enriches the process algebraic language with a symbolic representation of the (possibly infinite) state space of a process by means of state variables and formulas concerning these variables. μ-CRL uses a Concrete Invariant Corollary saying that if G is convergent and the parallel composition of C_is and *Spec* are solutions of G under the assumption of some invariant, then the two processes are equal in all states satisfying the invariant.

π-calculus-like Formalizations

First Extension Berger and Honda [5, 6] noticed that the π-calculus, as it was introduced by Milner, Parrow and Walker [42], did not suffice for the sound description of basic elements of distributed systems like message loss, timers, sites, site failure and persistence. And therefore extensions that clearly represented these phenomena were strongly needed to model distributed systems and offer useful reasoning frameworks. Thus, they proposed an extension to an asynchronous version of the π-calculus and examined its descriptive power for the description and correctness proof of the Two-Phase Commit Protocol. Such work is a considerable step forward towards the creation of a calculus in which it is possible to describe all the distributed systems. However, the Berger-Honda calculus would need some major extensions, since it still lacks of some important elements, one evident example is that the action of ”suspecting the crash of another process” is missing.

Adding Suspicions Recently, Francalanza and Hennessy [22] have designed a partial-failure process calculus to describe distributed algorithms, and they have proposed a methodology for proving correctness using fault-tolerance bisimulation techniques. More precisely, they have given a bisimulation-based proof for a very simple Consensus algorithm relying on perfect failure detectors. This work shows that it is undoubtedly feasible to use process calculi for describing and proving simple distributed algorithms. Unfortunately, real distributed algorithms cannot rely on perfect failure detectors, and the example proposed by Francalanza and Hennessy requires major revisions in order to be able to express such a minor change.

12.2.5 Petri Nets

Petri nets [47] are a model of computation for concurrent and distributed systems. A Petri net is composed of a set of *places*, holding some tokens, and a set of *transitions*, which fix how places can win or lose tokens. During the activation of a transition, which is the basic evolution step of the model, some of the places lose tokens while some other places win tokens. The number of tokens that is won or lost by a given place during the activation of a given transition is only function of the transition and the place, so it is the same for each activation of the transition. Of course a transition can be activated only if the places that are going to lose tokens during its activation hold enough tokens.

There exists also a variant, randomized Petri nets [56], created to formally represent randomized distributed algorithms. As opposed to original Petri nets, such variant has non-sequential semantics that can catch the representation of algorithms terminating with probability 1.

As for most formalisms based on a notion of reduction, the properties of a given Petri nets are usually derived from invariants, i.e. properties that are initially true and that are preserved by a reduction step. In the context of Petri nets, an invariant that states that a quantity computed from the number of tokens held by the different places is constant is called a *P-invariant*.

For our formalizations of Consensus algorithms, we used a different model where a set of guarded actions modify a global state. It seems that the model of Petri nets lies between the model we used and automata. Like automata Petri nets are compositional: sub-components of systems are represented as sub-Petri nets. Like in the model we used, Petri nets have a global mutable state that consists of the repartition of tokens between its places, and transitions of Petri nets are also guarded in the sense that they are only activable in certain contexts. Finally, it is interesting to remark that the assertional reasoning (or invariants) proof techniques apply to our formalization and to Petri Nets as well as to most other kinds of models.

12.2.6 Heard-Of Model

The heard-of model [14] (in short, HO model) is a mathematical framework for describing round-based distributed algorithms and the properties of their environment. The model is based on the simple observation that most round-based algorithms, and in particular algorithms for Consensus, respond to the two following principles: (1) the set of messages that are sent by a process in a round depends only on its current state, (2) the modification of a process's state from one round to the next depends only on its current state and on the messages it has received in the current round from other processes. As a consequence, the model focuses on the concept of round, rather than on time. The properties of the environment, such as the degree of synchrony and the distribution of process crashes in number and time, are described as a predicate on the set of processes that are *heard of* (from which a message has been received) by a given process at a given round.

At first glance, the HO model resembles our way of formalizing. Like in our model, the HO model takes a global view of a system of processes and see them as the vector (or configuration) of their individual states. And similarly, such configurations evolve by following the rules of a transition system. However, the similarities are only superficial since the HO model does not group in a configuration the states of all processes *at a given date* but the states of all processes *at a given round*. Since different processes can be in different rounds at the same time, this leads to a completely different view on the evolution of a system. Our formalization is already an abstraction of a real system, but the HO model goes actually further: it even abstracts over central phenomena like the life of a message on a communication link or the status (crashed or not) of a process.

The paper contains a description of a generalization of the Paxos algorithm in the HO model, together with a predicate on heard-of sets that is shown to enforce the termination of the algorithm. The advantage of the model, namely its very abstract view on the evolution of a system, is also its main drawback: it is not always obvious to see what a predicate on heard-of sets represents in the traditional and intuitive view (the one we choose) of processes sending messages on a channel, waiting for a response and potentially crashing.

12.3 Formalization and Verification Examples

One important example of application of μ-CRL was given by Fredlund, Groote and Korver [23] who presented a formal description and formal proofs of correctness of the Dolev, Klawe and Rodeh's leader election algorithm: a round based algorithm that has been designed for a network with a unidirectional ring topology where, in each round, every active process exchanges messages only with its neighbors and the number of electable processes decreases until the last process left declares itself the leader.

Felty and Stomp [20] presented the formalization and verification of cache coherence, a portion of the IEEE Scalable Coherent Interface used for specifying communication between multiprocessors in a shared memory model. The correctness proof is carried out for an arbitrary, finite number of processors and under the assumption that message order is preserved, the modeling is done through a program written in a guarded command language similar to UNITY, and the proof that the program satisfies its specifications is carried out with Linear Time Temporal Logic, making use of history variables. Structuring of the proof is achieved by means of auxiliary predicates.

IEEE standards have been the target of study of other formalizations. The IEEE 1394 high performance serial multimedia bus protocol allows several components to communicate with each other at high speed. Devillers, Griffioen, Romijn and Vaandrager [18] gave a formal model and verification of a leader election algorithm that forms the core of the tree identify phase of the physical layer of the 1394 protocol. The authors first describe both the algorithm and its specification in the I/O Automata model, then they use refinement mapping from the implementation to the specification to verify that, for any arbitrary tree topology, exactly one leader is elected. A large part of the verification has been checked mechanically with a verification system for higher-order logic, PVS [46]. Stoelinga and Vaandrager [53] have tackled another aspect of IEEE1394: the root contention protocol. This protocol involves both real time and randomization and it has to be run to find out what node of a tree is root. The verification of the algorithm is carried out in the probabilistic automaton model of Segala and Lynch [51] and the correctness of the protocol is proved by establishing a probabilistic simulation between the implementation and the specification, both probabilistic automata.

Kindler, Reisig, Völzer and Walter used Petri Nets to give a formalization of the echo algorithm, an algorithm used on finite connected network of agents to distribute data from an initiator to all other processes, thus creating a spanning tree of the graph. A variant of the Petri Nets, the Coloured Petri Nets, was also used to represent and evaluate the performance of CT in [52]

As we can see from the examples above, most of the formalization work that has been done up to now is about relatively simple algorithms (the Alternating Bit or the Leader Election protocols) and/or relatively simple settings (absence of process crashes, processes communicating only to their direct neighbors or through shared memory). While this is for sure a big improvement from the previous absence of rigor, still the available examples do not answer to the question whether the existing methods can be efficiently employed also to describe more complex algorithms (for example like Consensus) and in systems with more complex settings like failures and point-to-point unreliable communication. In fact, we observe that usually the description efficacy of a formalization is inversely proportional to the complexity of the system that one wants to represent, and very often, when the complexity increases, the readability of a formalization becomes close to null causing the formalization itself to become useless for the desired purposes.

Among the few formalizations of complex algorithms we are aware of, we cite Pogosyants, Segala and Lynch [48] who used Probabilistic I/O Automata as the basis for a formal presentation and proof of the randomized consensus algorithm of Aspnes and Herlihy [3]. They expressed the properties of the system in terms of sequences of external actions that the automata can provide (traces) and they carried out the proofs showing the validity of some invariants on the traces.

De Prisco, Lampson and Lynch [17] also used automata under the form of Clock General Timed Automaton models to give a new presentation of the Paxos algorithm. They decomposed the algorithm into several interacting components, gave a correctness proof of it and a fault tolerant analysis. It should be noticed that the proof uses automata composition and invariant assertion methods instead of the possibility mapping proposed by Lynch for general I/O Automata proofs. In their paper, we can find a proof of Termination of Paxos. However, it should be noticed that they perform their study in a partially synchronous system where the fixed upper bounds of message transmission time and of process speed are known (see 12.5.2) and that they augment the algorithm description with a module (STARTER) that is in charge of starting a new round only after the time bounds have been passed.

Gafni and Lamport [25] used TLA+ to describe the algorithm and the proofs of correctness of Disk Paxos, a variant of Paxos based on a network of processes and disks. This paper presents only specification and proof of safety property, with no interest for liveness.

Starting from Gafni and Lamport's work, Jaskelioff and Merz [31] achieved a machine-checked proof of correctness of the Disk Paxos algorithm using a translation of TLA+ in the Isabelle/HOL theorem prover.

Rushby [49] used PVS [46] to formalize and prove the Oral Messages algorithm proposed by Lamport, Shostak and Pease [38]. Young [58] chose the same algorithm to demonstrate that ACL2 [32] was as good as PVS for achieving the target. This led Young to argue that comparison of proof systems should not be made only with respect to small problems or to set of problems within a particular domain because they could be biased by cleverness in formalization and proofs.

12.4 Model Checking

A different approach from what we have seen above is given by model checking. Model checking is a technique for automatically verifying dynamic systems. It consists in checking whether a given structure (representing the system) is a model of a given logical formula (representing the specification). If the answer is negative, a detailed counter example is usually produced.

Typically, the structure is described through a graph where the nodes are the states of the system and the edges represent possible transitions which may alter the state. Each node has an associated set of atomic propositions representing the basic properties that hold at a point of execution. The analysis is performed by an exhaustive simulation of the model on all possible inputs. Model checking is computationally very expensive: the number of states of a system grows exponentially with the number of variables used to describe it, thus generating the so-called state-space explosion problem.

Little work has been done on verifying asynchronous distributed Consensus algorithms using model checking. This is mainly due to the fact that unbounded round numbers and unbounded messages quickly induce huge (infinite) state spaces. Among the successful trials, we remember the work of Tsuchiya and Schiper and of Kwiatkowska, Norman and Segala. Tsuchiya and Schiper [54] used the HO model to check the correctness of instances of Paxos having small number of processes. Given the constraint on the number of processes, this approach cannot provide a correctness proof for the general case, but checking is fully automated and is also useful for testing whether the algorithm is correctly designed. Kwiatkowska, Norman and Segala [34] checked the Aspnes and Herlihy randomized consensus algorithm [3]. In order to avoid the state-space explosion problem, they used the approach introduced by Pogosyants, Segala and Lynch [48], separating the algorithm into a non-probabilistic component and a probabilistic one. The non-probabilistic part was verified with the use of a proof assistant, while the coin-flipping protocol was verified using a model checker.

12.5 Auxiliary Structures and Additional Conditions

In this section, we present an overview of auxiliary structures that are often used in order to facilitate the reasoning for correctness proofs. Such structures improve the knowledge on the evolution of the system, usually without influencing the execution. We also report about some synchrony conditions that can be added to the system in order to overcome the FLP impossibility result and ease algorithm termination.

12.5.1 History Variables

History variables were mentioned for the first time by Clint in [15] to state and prove correctness criteria of programs and computations. The history consisted of a stream of values recording the evolution of some of the variables in the program. The correctness of the computation was expressed in terms of variables' final values and the relation of these values to the initial data. Later on, Owicki and Gries [45] used the concept of auxiliary variables in the scope of parallel programs to record the history of execution and to indicate which part of a program was executed at a certain time. The variables were called auxiliary because they were needed only for the proofs and not in the program itself (as opposed to Clint's history variables that were proper variables of the program). However, in the seventies, proving correctness of parallel programs meant showing that the properties claimed for one single process were not depending on other processes (they were interference-free). Therefore the concept of auxiliary variables is not applicable in our study, since we prove here the correctness of an algorithm involving communication and ”interference” between processes. The closest definition to the kind of history variables presented in this paper was given by Abadi and Lamport [1]. They used history variables as an augmentation of the state space, in such a way that the additional components recorded past information to be used in the proofs, but did not affect the behavior of the original state components. The concept of history can be found also when dealing with databases. For example, in [8], the term history indicates the order with which the operations of the transactions were executed relative to each other.

12.5.2 Partial Synchrony

Analyzing the role of asynchronism is distributed computing, and putting it in relation with the FLP result, Dwork Lynch and Stockmeyer [19] came up with the concept of *partial synchrony*, which lies between the cases of synchronous systems and asynchronous systems.

A synchronous system is characterized by the fact of having a known fixed upper bound Δ on the time required for a message to be sent from one process to another, and a known fixed upper bound Φ on the relative speed of different processes. For an asynchronous system, instead, no fixed upper bounds Δ and Φ exist. The authors propose two versions of partial synchrony:

- one where fixed bounds Δ and Φ actually exist, but they are not known a priori nor they can be determined through experiments;
- another one where bounds Δ and Φ are known, but they are only guaranteed to hold starting at some unknown time T.

In the first case, the problem is to design Consensus algorithms that work correctly in the partially synchronous system, regardless of the actual values of Δ and Φ. In the second case, the problem is to design Consensus algorithms that work correctly regardless of when time T occurs. In both cases the two parameters Δ and Φ can be combined to give all the possible degrees of asynchrony. For example, we can know everything about Φ, but Δ can be unknown (its value or its occurring time, depending on the case), or we can have no information on any of them. The authors thus propose some general fault-tolerant Consensus algorithms for various cases of partial synchrony and for various fault models. Moreover, they show that their proposed algorithms are optimal with respect to the number of faults tolerated.

Two aspects of this work should be pointed out here. The first one is that the second version of partial synchrony allows a generalization of the properties of quasi-reliable point-to-point communication and of failure detection (and indirectly also of reliable broadcasting). In fact, they are all depending on the fact that *eventually something good will happen*. The second one is that all the proposed protocols are strictly round based: every process in the system proceeds sequentially phase after phase, round after round. Thus this study is a generalization of algorithms like CT, but does not give many answers about algorithms non strictly sequential like Paxos. Although we believe that partial synchrony could help designing conditions for enforcing Paxos termination, no work has been done yet (to our knowledge) on this topic.

12.6 Our Opinion

As we have seen above, all the models proposed up to now have been efficient in describing some kind of problem. However, in our work, we have tried to put ourselves in the shoes of an hypothetical researcher in the Distributed Algorithm field and we have asked ourselves ”would we be ready to spend time learning a new language, new notations and new conventions only to be able to express our algorithm more formally?”. Our answer has been ”not really”. In fact, we think that both UNITY and μ-CRL are too theoretical, and therefore too cryptic, for somebody who is used to express ideas in pseudo-code. TLA+ requires instead quite a good knowledge of temporal logic in order to be appreciated. Furthermore, the indirect representation of transition rules through the use of logical formula in TLA+ is not very intuitive for a non-experienced reader. As for I/O Automata, if we consider the example [17] presented above, we see that the proofs are done in the same way as we did: providing a list of properties (invariants) of the executions and finally showing correctness. We therefore think it is a bit too demanding to learn the theory behind I/O Automata if then we do not fully exploit it for the proofs.

Thus, we suspect that the reason why so many distributed algorithms (and their correctness proofs) are still expressed in an informal way is that none of the existing methods meets the needs of researchers in the Distributed Algorithms field: clarity and simplicity without the constraint of learning new, complex theories that might not be known or understood by others. We believe that our formalization can help in this direction, since it is quite immediate to understand and, by forcing the designer to specify all the details of the system, it can also be an excellent starting point for anybody who wishes to formalize the algorithm into any of the other models we presented earlier.

12.6.1 Why We Do Not Recommend Process Calculi

When we decided to propose a new way of writing distributed algorithms, we thought that the use of process calculi was more than appropriate for this task: a well made calculus could describe the actions of each process, clearly stating what messages were required to take steps, and it could intuitively express the fact that the system was composed of processes that worked in parallel. Moreover, the fact that process calculi are equipped with formal operational semantics made us believe that proofs on top of them could be considered more formal than the pseudo-code based counterparts. The formalizations made by Berger and Honda [5, 6] gave us great hope in this sense.

Unfortunately, the Chandra-Toueg Consensus algorithm is far more complex than the Two Phase Commit Protocol, and while trying to use process calculi to describe it [44] we encountered such a number of problems that we started believing that was not the best way to go when wanting to formalize *complex* distributed algorithms. In fact, in such algorithms, processes often run concurrent threads and need to store quite rich state information. Moreover, usually only few state parameters are completely local: sometimes one or more state variables are shared among threads of the same process, while other state variables can contain message buffers that are shared among homologous threads of different processes. In process calculi, state variables are often modeled as parameters of process constants. However, this is no longer possible when state variables are to be shared across independent threads. The only solution is to have one thread holding the state while others get access to it by means of communication: in this way, reading and writing become explicit actions. While this is possible, it is not feasible in practice because it results in an outbreak of communication steps that make any subsequent formal reasoning extremely difficult.

Another problem with using process calculi is given by the fact that, usually, complex distributed algorithms are not built on top of a simple low-level message passing module, but are rather inserted in a layered architecture of several components, each providing and requiring specific communication services. Up to now, process calculi off the shelf do not offer any support of layered architectures. Process calculi usually offer a single hard-coded underlying service (typically: handshake message passing) that can be used to simulate any other service that is needed. However, if we do not wish to have a simulation, but a real service, we must extend process calculi with additional hard-coded communication mechanism, with the result of complicating the theory.

The last, but not least problem that we have encountered while trying to formalize distributed algorithms using process calculi is that, usually, correctness proofs for distributed algorithms require starting from a certain point in the execution and reason about what could and could not have happened before that point. If we use process calculi, we cannot simply use the information that is contained in the syntax of the process term because events of the past leave no traces on it. The solution to this is to either keep track of all the single events during an execution and search through all previous steps each time we want to verify whether something has happened in the execution or not, or we can equip the operational semantics with some global book-keeping data structure that logs all communication events. The first solution is too time consuming because it requires retracing the entire evolution of the execution every time we want to check if something has happened in the past. The second solution introduces a big gap between the semantics of the calculus and the semantics that is needed to express the evolution of the book-keeping structure, thus making the correctness proofs very hard and cryptical.

Chapter 13

Conclusion and Future Work

13.1 Conclusion

We have presented a new way of formalizing distributed Consensus algorithms using a global transition system generated by inference rules. As opposed to other previous methods, our proposal does not require any prior knowledge of temporal logics or automata, nor it requires the study of a new specific language. The formalization through transition rules is quite close to the traditional pseudo-code expression of distributed algorithms, thus resulting pretty intuitive to the users/readers. On the other hand, the high level of details required to describe algorithms through transition rules allows to overtake all problems of ambiguity and different interpretations introduced by the mere use of pseudo-code. In fact, since every component of the system needs to be specified before being used, the expression of the algorithms through transition rules forces the writer to deeply think and describe every module, every step of the algorithm and every interaction module(s)-algorithm, thus providing a rigorous specification. Our proposed formalization through transition rules can be seen both as an efficient way of expressing distributed algorithms, and proving their correctness with respect to the required properties, and as a first step towards more complex formalizations using TLA+, I/O Automata, μCRL and so on. In fact, our formalization simply aims to introduce a clear and unambiguous description of distributed algorithms which is the base for any other formalization, operation or implementation one might want to execute.

We have used our method to describe and prove the correctness of two distributed Consensus algorithms: the Chandra and Toueg and the Paxos algorithms. This exercise has allowed us to notice that, even though the algorithms look pretty different in their pseudo-code and natural language descriptions, they are both based on similar ideas and they execute very similar actions. This remark could be made by putting side by side the transition rules describing the algorithms and comparing their premises and effects. Also the correctness proofs could be made using the same structures and providing more or less the same lemmas and theorems for both algorithms. With the except of Termination.

In fact, while the Chandra-Toueg algorithm respects the Termination property, we proved that Paxos does not. This proof was possible because, using our formalization, we were able to find at least a run that does not terminate. The implications of this lack of termination are multiple, and span from questions on the correctness of real systems that are already including implementations of Paxos to questions whether it is possible to slightly modify the algorithm, or the required system settings, in order to guarantee its termination. But the most important question leads us to discuss whether Paxos can be considered as a Consensus algorithm or not. In fact all the definitions of Consensus given in literature lead us to believe that Termination is a fundamental property of the problem. Moreover, if we did not consider Termination as a strong requirement, the FLP result would not generate any trouble, since the failure of processes would not prevent any well designed algorithm to respect the safety properties of Agreement and Validity.

We think the answer to this question is still widely open. We were able to prove that, if the probability distribution of message delivery eventually becomes constant during the execution of the algorithm, then Paxos terminates with probability 1. We are still looking for evidence that our hypotheses are respected in all systems and we are also exploring other solutions.

13.2 Future work

From what we have seen in the previous Chapters and resumed above, we think that the most striking question that is left open is: ”how can we give conditions to guarantee the termination of Paxos?”. We believe that a possible answer could be found in the domain of *fairness*. There are already many definitions of fairness and many studies of applications of this concept to distributed algorithms. However, we are not aware of any work that has been done explicitly on Paxos. We believe that, in Paxos, the problem lies on the possibility for actions of being executed too late. For example, we can take the boomerang game and impose, as fairness condition, that if an infinity of boomerangs is launched, then an infinity of boomerangs comes back. We see that this is not enough to guarantee the termination of the algorithm. In fact, there is always the possibility that the boomerangs come back after the wizard has launched a new one. Boomerangs thus respect the fairness condition, but do not make the algorithm terminate. We are currently working on this problem and we believe it deserves deeper investigation from both the formal method and the distributed algorithm communities.

Another interesting study would be to search for (and make explicit) the relationship between probability, fairness and partial synchrony. It is obvious that these three domains are three ways of seeing the same problem under different points of view. Some work already exists that put in relation probability and fairness [30] and we think this topic should be investigated further because it could shed light on the problem of Termination.

Finally, we believe that with our formalizations of the Chandra-Toueg and the Paxos Consensus algorithms (and with their correctness arguments as well) we have provided sufficient detail to support a machine-checkable representation, and we are looking forward to that.

Appendix A

Chandra-Toueg Simulator

Starting from the description of the Chandra-Toueg algorithm in inference rules, it was possible to quickly implement a simulator [16] that allowed us to start playing with runs and helped us gaining an even deeper understanding of the algorithm behavior by sometimes providing counter examples to conjectures we expressed. We give here a short description of the implementation.

The execution is ruled by a simulation engine that is independent of the consensus algorithm. It is based on an agenda that is composed of entries. An entry in the agenda is a pair of a time and a set of actions that could be executed at that time. Time is discrete and represented by integers. The simulator proceeds in steps. The execution of a step consists in: looking for the entry with the smallest time and whose set of actions is not empty, setting this time as the current time of the simulation, randomly removing an action from the set and performing it. All actions that are not chosen in a step stay in the agenda and get a chance to be executed later. When executed, actions can insert new entries in the agenda. The time of these new entries is specified relatively to the current time of the simulation.

The simulator is parameterized by a set of values (number of processes, probability of crash, message delivering delays, etc). The implementation uses default values for theses parameters (except for the number of processes that must be passed on the command line), but the user can change these values by editing the source code. Examples of such parameters are:

prob(QPP_Send): probability for a QPP-message to be sent when the sender or the target of the message is an incorrect process.

prob(RBC_Correct_Deliver): probability for a RBC-message to be delivered to all correct processes when the sender is an incorrect one.

max_delay(QPP_Deliver): maximum delay for a QPP-message in transit

that must eventually been delivered to be actually delivered.
max_delay(Crash) = maximum delay for an incorrect process to crash.
max_time(Suspect) = maximum suspicion time for the immortal process.

The number of processes n is passed on the command line. The number p of correct processes is chosen randomly before running the simulation, and there must of course be a majority of correct processes. The "immortal" correct process is chosen randomly among the correct processes. The suspicion time, i.e. the period in which all processes including the immortal one can be suspected, is also chosen randomly.

Each process gets periodically activated and each time it does so, it tries to execute a rule ((WHILE), (PHS-1), (PHS-2), (PHS-3-TRUST), (PHS-3-SUSPECT), (PHS-4-FAIL) or (PHS-4-SUCCESS)). Then, it goes to sleep again until being reactivated after a delay randomly chosen.

The set of values contains as many elements as the number of processes. The initial belief of each process is chosen randomly in that set.

Each time a message becomes ready to be sent (outgoing) from a correct process to a correct process (through rules (PHS-1), (PHS-2), (PHS-3-TRUST), (PHS-3-SUSPECT)) rule (QPP_SND) is applied by the source process after a randomly chosen delay. Each time the rule (QPP_SND) is applied on a message from a correct process to a correct process, the rule (QPP_DELIVER) is applied by the target process after a randomly chosen delay. Each time a message becomes ready to be sent (outgoing) from a process to another (through rules (PHS-1), (PHS-2), (PHS-3-TRUST), (PHS-3-SUSPECT)) and at least one of them is not correct, there is a probability prob(QPP_Send) for the message to be sent. If it is decided that the message will be sent, (QPP_SND) is applied by the source process after a randomly chosen delay. Each time the rule (QPP_SND) is applied on a message from a process to another where at least one of them is not correct, there is a probability prob(QPP_Deliver) for the message of being delivered. If it is decided that the message will be delivered, (QPP_DELIVER) is applied by the target process after a randomly chosen delay. A similar implementation is done for Reliable Broadcast messages.

The rule (PHS-3-SUSPECT) cannot be applied in case the coordinator is the immortal process and the simulation time is greater than the suspicion time. The rule can be applied in all other cases provided its premises are satisfied.

For each incorrect process, the rule (CRASH) is applied after a delay chosen randomly between 0 and max_delay(Crash).

A possible evolution of such work could be to implement the same also for Paxos and compare the speed and efficiency of the two algorithms by letting them run in the same settings.

Bibliography

[1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991. A preliminary version appeared in the proceedings of LICS'88.

[2] M. K. Aguilera and S. Toueg. Failure detection and randomization: A hybrid approach to solve consensus. *SIAM Journal on Computing*, 28(3):890–903, 1999.

[3] J. Aspnes and M. Herlihy. Fast randomized consensus using shared memory. *Journal of Algorithms*, 11(3):441–461, 1990.

[4] M. Ben-Or. Another advantage of free choice (extended abstract): Completely asynchronous agreement protocols. In *PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 27–30, New York, NY, USA, 1983. ACM Press.

[5] M. Berger. *Towards Abstractions for Distributed Systems*. PhD thesis, Imperial College, Dept. of Computing, 2002.

[6] M. Berger and K. Honda. The two-phase commitment protocol in an extended pi-calculus. In Luca Aceto and Björn Victor, editors, *EXPRESS '00: Proceedings of the 7th International Workshop on Expressiveness in Concurrency (State College, USA, August 21, 2000)*, volume 39.1 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science Publishers, 2000.

[7] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37(1):77–121, ???? 1985.

[8] P. A. Bernstein, V. Hadzilacos, and N. Goodman. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley, 1987.

[9] D. Bünzli, R. Fuzzati, S. Mena, U. Nestmann, O. Rütti, A. Schiper, and P. T. Wojciechowski. *Dependable Systems: Software, Computing,*

Networks, volume 4028 of *Lecture Notes in Computer Science*, chapter Advances in the Design and Implementation of Group Communication Middleware, pages 172–194. Springer, 2006.

[10] T. D. Chandra, R. Griesemer, and J. Redstone. Paxos made live - an engineering perspective (2006 invited talk). In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing, 2007*, pages 398–407. ACM Press, 2007.

[11] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, 1996.

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

[13] K. M. Chandy and J. Misra. *Parallel Programming Design. A Foundation.* Addison Wesley, Reading MA, 1988.

[14] B. Charron-Bost and A. Schiper. The Heard-Of Model: Computing in Distributed Systems with Benign Failures. Technical report, 2007. Replaces TR-2006: The Heard-Of Model: Unifying all Benign Failures.

[15] M. Clint. Program proving: Coroutines. *Acta Informatica*, 2:50–63, 1973.

[16] V. Cremet and R. Fuzzati. Chandra-toueg consensus algorithm simulator. www.epfl.ch/∼blackbal/Thesis/Simulator, 2005.

[17] R. De Prisco, B. Lampson, and N. Lynch. Revisiting the paxos algorithm. In M. Mavronicolas and Ph. Tsigas, editors, *Distributed Algorithms; Proceedings of the 11th International Workshop, WDAG'97, Saarbrücken, Germany*, volume 1320 of *Lecture Notes in Computer Science*, pages 111–125. Springer-Verlag, 1997. September.

[18] M. Devillers, W. O. D. Griffioen, J. Romijn, and F. W. Vaandrager. Verification of a leader election protocol: Formal methods applied to ieee 1394. *Formal Methods in System Design*, 16(3):307–320, 2000.

[19] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the Association for Computing Machinery*, 35(2):288–323, April 1988.

[20] A. Felty and F. Stomp. A correctness proof of a cache coherence protocol. In *Compass'96: Eleventh Annual Conference on Computer Assurance*,

page 128, Gaithersburg, Maryland, 1996. National Institute of Standards and Technology.

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

[22] A. Francalanza and M. Hennessy. A fault tolerance bisimulation proof for consensus. In *Proceedings of the 16th European Symposium on Programming ESOP '07*. Springer-Verlag, 2007.

[23] L. Fredlund, J. F. Groote, and H. Korver. Formal verification of a leader election protocol in process algebra. *Theoretical Computer Science*, 177(2):459–486, 1997.

[24] R. Fuzzati, M. Merro, and U. Nestmann. Distributed consensus, revisited. *Acta Informatica*, 44(6), October 2007.

[25] E. Gafni and L. Lamport. Disk paxos. *Distributed Computing*, 16(1):1–20, 2003.

[26] J. F. Groote and A. Ponse. Proof theory for μCRL: A language for processes with data. In *Proceedings of the International Workshop on Semantics of Specification Languages (SoSL)*, pages 232–251, London, UK, 1994. Springer-Verlag. Also appeared as: Technical Report CS-R9138, CWI, Amsterdam, 1991.

[27] J. F. Groote and A. Ponse. The syntax and semantics of μcrl. In *Algebra of Communicating Processes, Workshops in Computing*, pages 26–62. ???, 1994.

[28] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 13(1):124–149, 1991.

[29] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[30] M. Jaeger. Fairness, computable fairness and randomness. In *Proceedings of the 2nd International Workshop on Probabilistic Methods in Verification (PROBMIV99)*, 1999.

[31] M. Jaskelioff and S. Merz. Proving the correctness of disk paxos. http://afp.sourceforge.net/entries/DiskPaxos.shtml, jun 2005.

[32] M. Kaufmann and J S. Moore. Acl2. http://www.cs.utexas.edu/users/moore/acl2/.

[33] D. K. Kaynar, N. Lynch, R. Segala, and F. Vaandrager. Timed I/O automata: A mathematical framework for modeling and analyzing real-time systems. In *RTSS 2003: The 24th IEEE International Real-Time Systems Symposium, Cancun, Mexico*, pages 166–177, 2003.

[34] M. Kwiatkowska, G. Norman, and R. Segala. Automated verification of a randomized distributed consensus protocol using cadence SMV and PRISM. *Lecture Notes in Computer Science*, 2102:194–207, 2001.

[35] L. Lamport. The part-time parliament. *ACM Trans. Comput. Syst.*, 16(2):133–169, 1998.

[36] L. Lamport. Paxos made simple. *SIGACT News*, 32(4):18–25, 2001.

[37] L. Lamport. *Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers.* Pearson Education, 2002.

[38] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[39] N. Lynch, R. Segala, and F. Vaandraager. Hybrid I/O automata. *Information and Computation*, 185(1):105–157, 2003.

[40] N. A. Lynch and M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. In *PODC*, pages 137–151, 1987.

[41] R. Milner. *A Calculus of Communicating Systems.* Springer-Verlag, 1980.

[42] R. Milner, J. Parrow, and D. Walker. A calculus of mobile processes, part I/II. *Information and Computation*, 100:1–77, September 1992.

[43] U. Nestmann and R. Fuzzati. Unreliable failure detectors via operational semantics. In Vijay A. Saraswat, editor, *Proceedings of ASIAN 2003*, volume 2896 of *Lecture Notes in Computer Science*, pages 54–71. Springer-Verlag, December 2003.

[44] U. Nestmann, R. Fuzzati, and M. Merro. Modeling consensus in a process calculus. In *CONCUR*, pages 393–407, 2003.

[45] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs i. *Acta Informatica*, 6(4):319–340, December 1976.

[46] S. Owre, J. M. Rushby, , and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*, pages 748–752, Saratoga, NY, jun 1992. Springer-Verlag.

[47] C. A. Petri. *Kommunikation mit Automaten.* PhD thesis, Rheinische Friedrich-Wilhelms-Universität Bonn, 1962.

[48] A. Pogosyants, R. Segala, and N. Lynch. Verification of the randomized consensus algorithm of aspnes and herlihy: a case study. *Distributed Computing*, 13(3):155–186, 2000.

[49] J. Rushby. Formal verification of an oral messages algorithm for interactive consistency. Technical Report SRI-CSL-92-1, Computer Science Laboratory, SRI International, Menlo Park, CA, jul 1992. Also available as NASA Contractor Report 189704, October 1992.

[50] R. Segala. *Modeling and Verification of Randomized Distributed Real-Time Systems.* PhD thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1995.

[51] R. Segala and N. Lynch. Probabilistic simulations for probabilistic processes. *Nordic Journal of Computing*, 2(2):250–273, 1995.

[52] N. Sergent. Performance evaluation of a consensus algorithm with petri nets. In *PNPM '97: Proceedings of the 6th International Workshop on Petri Nets and Performance Models*, page 143, Washington, DC, USA, 1997. IEEE Computer Society.

[53] M. Stoelinga and F. Vaandrager. Root contention in IEEE 1394. *Lecture Notes in Computer Science*, 1601:53–74, 1999.

[54] T. Tsuchiya and A. Schiper. Model checking of consensus algorithms. In *26th IEEE International Symposium on Reliable Distributed Systems (SRDS 2007)*, pages 137–148. IEEE Computer Society, 2007.

[55] M. R. Tuttle. Hierarchical correctness proofs for distributed algorithms. Master's thesis, Laboratory for Computer Science, Massachusetts Institute of Technology, 1987.

[56] H. Völzer. Randomized non-sequential processes. In *CONCUR*, volume 2154 of *Lecture Notes in Computer Science*, pages 184–201. Springer-Verlag, 2001.

[57] S. Wu, S. A. Smolka, and E. W. Stark. Composition and behaviors of probabilistic I/O automata. *Theoretical Computer Science*, 176(1–2):1–38, 1997.

[58] W. D. Young. Comparing verification systems: Interactive consistency in ACL2. *IEEE Trans. Softw. Eng.*, 23(4):214–223, 1997.

Curriculum Vitæ

Personal Data

Name	Rachele Fuzzati
Nationality	Italian
Date of Birth	August 19th, 1974
Place of Birth	Ferrara, Italy

Education

2002	2007	Ph.D. Programming Methods Laboratory (LAMP), EPFL, Switzerland
2001	2002	Pre-Doctoral School, EPFL, Switzerland
1993	2000	B.Sc. and Master in Electronic Engineering (Laurea in Ingegneria Elettronica), Università degli Studi di Ferrara, Italy
1988	1993	Liceo Scientifico ”Roiti”, Ferrara, Italy

Professional Experience, Internships

2001	Hewlett-Packard Labs Bristol
2000	Hewlett-Packard Labs Bristol