Round-Based Consensus Algorithms, Predicate Implementations and Quantitative Analysis

THÈSE N^{O} 4839 (2011)

PRÉSENTÉE LE 22 FÉVRIER 2011

À LA FACULTÉ INFORMATIQUE ET COMMUNICATIONS
LABORATOIRE DE SYSTÈMES RÉPARTIS
PROGRAMME DOCTORAL EN INFORMATIQUE, COMMUNICATIONS ET INFORMATION

ÉCOLE POLYTECHNIQUE FÉDÉRALE DE LAUSANNE

POUR L'OBTENTION DU GRADE DE DOCTEUR ÈS SCIENCES

PAR

Fatemeh BORRAN

acceptée sur proposition du jury:

Prof. D. Thalmann, président du jury
Prof. A. Schiper, directeur de thèse
Dr C. Cachin, rapporteur
Prof. R. Guerraoui, rapporteur
Prof. N. Suri, rapporteur

ÉCOLE POLYTECHNIQUE
FÉDÉRALE DE LAUSANNE

Suisse
2011

Abstract

Fault-tolerant computing is the art and science of building computer systems that continue to operate normally in the presence of faults. The fault tolerance field covers a wide spectrum of research area ranging from computer hardware to computer software. A common approach to obtain a fault-tolerant system is using software replication. However, maintaining the state of the replicas consistent is not an easy task, even though the understanding of the problems related to replication has significantly evolved over the past thirty years.

Consensus is a fundamental building block to provide consistency in any fault-tolerant distributed system. A large number of algorithms have been proposed to solve the consensus problem in different systems. The efficiency of several consensus algorithms has been studied theoretically and practically. A common metric to evaluate the performance of consensus algorithms is the number of communication steps or the number of rounds (in round-based algorithms) for deciding. A large amount of improvements to consensus algorithms have been proposed to reduce this number under different assumptions, e.g., nice runs. However, the efficiency expressed in terms of number of rounds does not predict the time it takes to decide (including the time needed by the system to stabilize or not).

Following this idea, the thesis investigates the round model abstraction to represent consensus algorithms, with benign and Byzantine faults, in a concise and modular way. The goal of the thesis is first to decouple the consensus algorithm from irrelevant details of implementations, such as synchronization, then study different possible implementations for a given consensus algorithm, and finally propose a more general analytical analysis for different consensus algorithms.

The first part of the thesis considers the round-based consensus algorithms with benign faults. In this context, the round model allowed us to separate the consensus algorithms from the round implementation, to propose different round implementations, to improve existing round implementations by making them swift, and to provide quantitative analysis of different algorithms.

The second part of the thesis considers the round-based consensus algorithms with Byzantine faults. In this context, there is a gap between theoretical consensus algorithms and practical Byzantine fault-tolerant protocols. The round model allowed us to fill the gap by better understanding existing protocols, and enabled us to express existing protocols in a simple and modular way, to obtain simplified proofs, to discover new protocols such as decentralized (non leader-based) algorithms, and finally to perform precise timing analysis to compare different algorithms.

The last part of the thesis shows, as an example, how a round-based consensus algorithm that tolerates benign faults can be extended to wireless mobile ad hoc networks using an adequate communication layer. We have validated our implementation by running simulations in single hop and multi-hop wireless networks.

Keywords: distributed algorithms, fault tolerance, consensus problem, partial synchrony, heard-of model, round model, round-based algorithms, quantitative analysis, swift algorithms, Byzantine consensus, leader-based algorithms, decentralized algorithms, wireless ad hoc networks.

Résumé

La tolérance aux fautes est l'art et la science de la construction de systèmes informatiques qui continuent à fonctionner normalement en présence de fautes. Le champ de la tolérance aux fautes est large, et couvre différents domaines de recherche allant du matériel informatique au logiciel. La réplication est une des techniques standard de tolérance aux fautes. Toutefois, maintenir cohérent l'état des répliques n'est pas une tâche aisée, même si la compréhension des problèmes liés à la réplication a progressée de manière significative durant les trentes dernières années.

Le problème du consensus est un élément fondamental pour assurer la cohérence des données dans un système réparti tolérant aux fautes. Un grand nombre d'algorithmes ont été proposés pour résoudre ce problème dans différents modèles de systèmes. L'efficacité des algorithmes de consensus a été étudié théoriquement et pratiquement. Une mesure commune pour évaluer la performance de ces algorithmes est le nombre d'étapes de communication ou le nombre de rondes (dans le cas d'algorithmes en rondes) pour décider. Une grande quantité d'améliorations aux algorithmes de consensus ont été proposés pour réduire ce nombre dans le cas de différentes hypothèses, tels par exemple les "*nice runs*". Cependant, le nombre de rondes ne permet pas de prévoir le temps nécssaire pour décider (y compris le temps nécessaire par le système pour stabiliser ou pas).

Motivée par cette observation, la thèse étudie le modèle de ronde pour exprimer les algorithmes de consensus, que ce soit avec des fautes bénignes ou Byzantines, sous une forme concise et modulaire. L'objectif de la thèse est d'abord de découpler l'algorithme de consensus des détails de mises en œuvre, comme la synchronisation, puis d'étudier différentes implémentations possibles pour un algorithme de consensus donné, et enfin de proposer une analyse analytique plus générale de différents algorithmes de consensus.

La première partie de la thèse considère les algorithmes de consensus en rondes avec des fautes bénignes. Dans ce contexte, le modèle de ronde nous a permis de séparer les algorithmes de consensus de la mise en œuvre des rondes, de proposer différentes implémentations de rondes, d'améliorer les implémentations de rondes existantes en les rendant "*swift*", et de fournir une analyse quantitative de différents algorithmes.

La deuxième partie de la thèse considère les algorithmes de consensus en rondes avec fautes Byzantines. Dans ce contexte, il existe un écart entre d'une part les algorithmes de consensus théoriques, et d'autre part les protocoles pratiques tolérants les fautes Byzantines. Le modèle de ronde nous a permis de réduire cet écart grâce à une meilleure compréhension des protocoles pratiques existants. Il nous a permis d'exprimer des protocoles existants de manière simple et modulaire, d'obtenir des preuves simplifiées, de découvrir de nouveaux protocoles tels les algorithmes décentralisés (sans leader), et enfin d'effectuer des analyses temporelles précises pour comparer différents algorithmes.

La dernière partie de la thèse montre, à titre d'exemple, comment un algorithme de consensus en rondes tolérant les fautes bénignes peut être étendu aux réseaux mobiles ad hoc sans fil en utilisant une couche de communication adéquate. Nous avons validé cette approche par des simulations dans un réseau sans fil à un hop et à plusieurs hops.

Mots-clés: algorithmes distribués, tolérance aux fautes, problème de consensus, système partiellement synchrone, modèle "*heard-of*", modèle en rondes, algorithmes en rondes, analyse quantitative, algorithmes "*swift*", consensus Byzantin, algorithmes avec leader, algorithmes décentralisés, réseau ad hoc sans fil.

Acknowledgments

It is a real pleasure to thank those people who contributed in one way or the other to the success of this work and made this thesis possible.

First and foremost, I would like to express my special thanks and appreciation to my thesis supervisor, Professor André Schiper, for providing me with the opportunity to work in the research area of distributed computing, and for his encouragement, support and confidence in all levels. It was truly a privilege completing this thesis under his guidance.

I would also like to express my gratitude to the members of my jury, Dr. Christian Cachin, Professor Rachid Guerraoui, and Professor Neeraj Suri who took time to read and examine my thesis and provided me with valuable comments, as well as the president of the jury, Professor Daniel Thalmann.

Furthermost, I am grateful to all my colleagues at the Distributed Systems Laboratory (LSR): Nuno Santos and Dr. Martin Hutle, who coauthored most of the work presented in this thesis and for the unforgettable trip that we had to Alaska, as well as Zarko Milosevic, Omid Shahmirzadi, and Darko Petrovic. Many thanks to France Faille for helping me with all the administrative tasks.

I would also warmly thank former members of LSR with whom I had a chance to work shortly, specially Dr. David Cavin and Dr. Yoav Sasson, who supervised my first project at LSR and introduced me at LSR, as well as Cendrine Favez, Dr. Olivier Rütti, Dr. Richard Ekwall, and Dr. Sergio Mena.

I warmly thank all my Swiss friends for helping me integrating in Switzerland and giving me a good reason to learn French, and all my Iranian friends for all the memorable moments that we shared together.

I express my deepest gratitude to my whole family for their unconditional support and continuous encouragement. Finally, my warmest thanks belong to Hooman, for being such a wonderful husband, and for his patience, love, and support. Without his encouragement and understanding it would have been impossible for me to finish this thesis. A special thought goes to the baby that we are eagerly expecting.

The research presented in this thesis was partly funded by the Swiss National Science Foundation under grant number 200021-111701. I thankfully acknowledge this support.

Contents

Part II Byzantine Faults

Part III Wireless Networks

List of Figures

List of Tables

Chapter 1

Introduction

"Whatever can go wrong will go wrong at the worst possible time and in the worst possible way."

Murphy's law

1.1 Thesis Context

Availability and *reliability* have become increasingly important in today's computer dependent world [Hen99]. Availability is defined as the property that a system is ready to be used immediately. Reliability refers to the property that a system can run continuously without failure. In many applications where computers are used, outage or malfunction is expensive or unacceptable. Just imagine that the database system of your banking account is not accessible when you have an important payment, or the computer system in an aircraft is malfunctioning. You would certainly change your mind about opening a banking account or taking a plane. To achieve the needed availability and reliability, *fault-tolerant* systems are required.

Fault-tolerance using replication One way to implement a fault-tolerant service is by using multiple servers that may fail *independently.* The state of the service is *replicated* at these servers, and updates are coordinated so that even when a subset of servers fail, the service remains available. One approach for replication management is the *state machine replication* or *active replication*, which has no centralized control [Lam78, Lam84, Sch90, Sch93]. In this approach, replica coordination and consistency are ensured by enforcing all replicas to receive and process the same sequence of requests. This condition can be achieved by a broadcast primitive called *atomic broadcast*, which guarantees that all correct processes deliver the same messages in the same order [HT93, DSU04]. Atomic broadcast is a problem that involves achieving some sort of *agreement* – namely agreement on the sequence of messages delivered by correct processes – in a fault-tolerant manner, and so has a common flavor with the *consensus* problem [Fis83]. Indeed, from the solvability point of view, consensus and atomic broadcast are equivalent in *asynchronous* systems with *crash failures* and also in asynchronous systems with *arbitrary faults* [CT96].

Consensus problem In the consensus problem, each process starts with an initial value from a fixed set V, and must eventually reach a common and irrevocable decision from V. Consensus can be easy or difficult to achieve depending on the communication system (synchronous or asynchronous) and the failure assumptions. In the famous paper [FLP85], Fischer, Lynch and Paterson showed the *impossibility* of *deterministic* consensus among two or more processes in an asynchronous distributed system, in the presence of crashes (so called FLP impossibility result). Since then, the consensus problem has been examined under many different synchrony and failure assumptions. For example, Pease, Shostak and Lamport [PSL80], among others, showed that consensus cannot be achieved in a synchronous environment if even one third of the processes are *maliciously* faulty – that is, if they act in a way that simulates an agent that tries to make the other processors make inconsistent decisions. Another result establishes that in a synchronous distributed system in which messages can be dropped, consensus is not possible even if none of the processes fail [Lyn96].

Following FLP impossibility result, several system models have been proposed, mainly the *partially synchronous systems* [DDS87, DLS88], and the asynchronous system augmented with *failure detectors* [CT96].

The failure detector model was really successful for the limited *crash-stop* model. The model allowed to identify the weakest failure detector that can be used to solve consensus [CHT96]. However, the efforts to extend the notion of failure detectors to more general models, such as crash-recovery model [ACT98], and arbitrary faults [DS98], were less successful.

On the other hand, the partially synchronous model was a more general approach, with the ability to consider process crash-recovery and arbitrary failures, but provides too low level abstraction to express consensus algorithms. It is indeed useful to provide higher level abstractions for expressing consensus algorithms.

Few years later, Gafni [Gaf98] has extended the round-based model to abstract away the implementation of the communication between processes, being shared memory or message-passing. The properties of the communication mechanisms and system guarantees are captured as a whole by a single module that is called *Round by Round Failure Detector* (for short *RRFD*) module. The module unifies the synchrony degree and the failure model in the same abstract entity. However, the RRFD model and its general extension GIRAF [KS06] consider process failure as with failure detectors, and ignore transient link failures. Santoro and Widmayer [SW89, SW05, SW07] have introduced the *transmission fault* model. A transmission fault can be due to a process failure as well as link failure.

Consensus and the HO model Following the ideas of RRFD model and transmission faults, the recently proposed *Heard-Of* model (*HO* for short) encapsulates the notion of faulty component (process or link) in a same *transmission fault*, and proposes *communication predicates* that provide a better abstraction to solve consensus [CBS09]. In the first and last part of the thesis the HO model is used to solve the consensus problem.

The HO model handles benign failures, being static or dynamic, permanent or transient, in a unified framework. In fact, the model can naturally represent link failures, contrary to models with failure detectors [CT96, Gaf98]. Another feature of the HO model is that contrary to the random model [BO83, Rab83] or the failure detector model, there is no notion of "augmenting" asynchronous systems with external devices (oracles) that processes may query: the communication predicate corresponding to an HO system is an integral part of the model and should be rather seen as defining the *environment*. In terms of implementation, the HO abstraction can be supported by the messages sent by the consensus algorithm (only few additional messages are added, depending on the communication predicate).

In the HO model, the conditions that ensure the safety and the liveness of the algorithms are expressed as predicates on rounds. The distributed consensus algorithms represented in this new model are simple and elegant. The abstraction provided with this model facilitates the proofs. An algorithm expressed in the HO model remains the same for reliable links, unreliable links, crash failures and send-omission failures. For solving consensus, contrary to the failure detector model, the HO model does not require some property to hold forever. Using the HO model, it is possible to quantify the time that a given algorithm takes to solve consensus. This leads to better understanding of consensus algorithms and provides meaningful comparison results.

Consensus and Byzantine faults Contrary to the partially synchronous systems, failure detectors cannot handle Byzantine (arbitrary) faults. The reason is that the definition of a Byzantine behavior is related to an algorithm: It is impossible to achieve a complete separation of failure detectors from the algorithm using them. To overcome this problem, the notion of *muteness detectors* has been suggested [DS97, DS98, KMMS97, BHR00]. However, it is not clear what system model could allow the implementation of muteness detectors, which is an inherent limitation of the approach.

The HO model is not limited to deal only with benign faults. It can be extended to handle Byzantine faults [BH09] or *value faults* [BCBG$^+$07] (with value faults messages may be corrupted, *i.e.*, at any round r, the message received by process q from process p may be different than the message that p has sent to q). The algorithms presented in those papers are simple and concise comparing to the existing protocols. However, they require strong predicates for safety or liveness. For instance, [BH09] require (Byzantine) processes to behave correctly for some time. Note that these algorithms are not comparable with existing Byzantine fault-tolerant algorithms.

The second part of the thesis considers the basic round model instead of the HO model for expressing consensus algorithms that tolerate Byzantine faults. The round model abstraction allowed us to better understanding of the existing Byzantine protocols, define modular abstractions and propose simple Byzantine consensus algorithms. It also enables us to perform an exact and precise analytical analysis of the algorithms, figure out the best and worst case results, and provide a rigorous comparison of different algorithms.

Consensus and MANETs A *mobile ad-hoc network (MANET)* consists of mobile hosts equipped with wireless communication devices. The transmission of a mobile host is received by all hosts within its transmission range, due to the broadcast nature of wireless communication and omni-directional antennae. If two wireless hosts are out of their transmission range, other mobile hosts located between them can forward their messages. Due to the mobility of wireless hosts, each host needs to be able to route messages; since no statically established infrastructure or centralized administration is available. The mobile hosts can move arbitrarily and can be turned on or off without notifying other hosts. The mobility and autonomy introduces a dynamic topology. Ad hoc networks can also benefit from replicating services across nodes in order to increase their availability.

Consensus algorithms that use failure detectors are constructed on top of the *reliable links*. In fact either the system must provide reliable links, or reliable links need to be implemented on top of the unreliable system links. However, reliable links cannot be provided for free in all environments. For instance, providing reliable broadcast in MANETs is quite challenging [SCS02, MCS+06, KS07]. In general, most abstractions inherited from the wired networks are usually not appropriate to MANETs.

The last part of the thesis considers consensus problem for MANETs with benign faults. We have noticed that the HO model is an appropriate model to extend consensus algorithms from classical networks to MANETs, since the model can easily handle packet loss and packet collision. The essential difference between one-hop and multi-hop algorithms can be encapsulated within an adequate communication layer. Finally, the required connectivity of the network graph can be enclosed in a liveness predicate.

1.2 Thesis Contribution

The thesis has the following main contributions:

Quantitative analysis of consensus algorithms Consensus is one of the key problems in fault-tolerant distributed computing. A very popular model for solving consensus with benign faults is failure detector model defined by Chandra and Toueg. However, the failure detector model has some limitations as already discussed. We consider instead the HO model, and discuss several implementations of communication predicates in a system that alternates between good periods and bad periods. This approach allows us to quantify the required length of a good period to solve one or more instances of consensus. With our results, we can make several interesting observations such as the number of rounds is in general not a good prediction for the time needed to solve consensus.

Swift algorithms for repeated consensus The classical round-based model implementation has a main drawback. Mainly, it does not allow making progress at the speed of the system. We introduce the notion of a *swift algorithm*. Informally, an algorithm that solves a repeated problem is swift, if in a partial synchronous run of this algorithm, eventually no timeout expires, *i.e.*, the algorithm execution proceeds with the actual speed of the network. This definition differs from other efficiency criteria for partial synchronous systems.

Furthermore, we show that the notion of swiftness explains the reason why failure detector based algorithms are typically more efficient than round-based algorithms, since the former are naturally swift while the later are naturally non-swift. We show that this is not an inherent difference between the models, and provide a round implementation that is swift, therefore performing similarly to failure detector algorithms while maintaining the advantages of the round model.

Decentralized Byzantine consensus algorithms We also consider the consensus problem in a partially synchronous system with Byzantine faults. It turns out that, in the partially synchronous system, all deterministic algorithms that solve consensus with Byzantine faults are leader-based. This is not the case of benign faults, which raises the following fundamental question: is it possible to design a deterministic Byzantine consensus algorithm for a partially synchronous system that is not leader-based? We give a positive answer to this question, and present a decentralized (non leader-based) algorithm that is resilient-optimal and signature-free.

We have designed our decentralized algorithm using a methodology that consists of extending a synchronous consensus algorithm to a partially synchronous system using an asynchronous algorithm. While the asynchronous algorithm ensures safety in all rounds, the synchronous algorithm provides liveness in periods of synchrony.

Timing analysis of leader-based and decentralized Byzantine consensus algorithms We compare two leader-based and decentralized algorithms for Byzantine consensus with strong validity in an analytical way. We show that for the algorithms we analyzed, in most cases, the decentralized variant of the algorithm shows a better worst-case execution time. Moreover, for the practically relevant case $t \leq 2$ (t is the maximum number of Byzantine processes), this worst-case execution time is even at least as good as the execution time of the leader-based algorithms in fault-free runs.

Extending Paxos/LastVoting for wireless ad hoc networks Solving consensus in wireless ad hoc networks has started to be addressed in several papers. Most of these papers adopt system models similar to those developed for wired networks. These models are focused towards node failures while ignoring link failures, and thus are poorly suited for wireless ad hoc networks. The recently proposed HO model does not have this drawback. We show that an existing algorithm expressed in the HO round model can be used for multi-hop wireless ad hoc networks, if extended with an adequate communication layer. The description of the communication layer is augmented with simulation results that validate the feasibility of our approach and provide better understanding of the behavior of the wireless environment.

1.3 Thesis Outline

The thesis is organized as following:

Definitions and background Chapter 2 specifies the different system models considered in this thesis and defines concepts such as the HO model and the round model used throughout the thesis. This chapter also includes formal definitions of the problems addressed in this thesis as a background.

Benign faults The first part of the thesis considers consensus problem with benign faults. Chapter 3 presents a quantitative analysis of different round-based consensus algorithms. Chapter 4 shows that round-based algorithms can also make progress at the speed of the system. It introduces the notion of a swift algorithm and presents a round-based consensus algorithm that is swift.

Byzantine faults The second part of the thesis considers consensus with Byzantine faults. Chapter 5 presents two novel decentralized deterministic consensus algorithms that tolerates Byzantine faults, without using digital signatures. Chapter 6 compares the two variants of the Byzantine consensus algorithms presented in Chapter 5, leader-based variant and decentralized variant, in an analytical way, and shows that the decentralized variant outperforms the leader-based variant in the worst case.

Wireless networks The last part of the thesis extends a round-based consensus algorithm, that tolerates benign faults, for wireless ad hoc networks using an adequate communication layer. The algorithm is evaluated by simulation in single-hop and multi-hop wireless networks.

Conclusion Chapter 8 summarizes the main results of this work and identifies areas for future research.

Chapter 2

Definitions and Background

The current chapter describes precisely all the assumptions and definitions about the system considered throughout the thesis. We start with the definition of the system model and synchrony assumptions. Then we describe the fault model, including process failure and link failure. Finally we give the definitions for the problems addressed in this thesis as a background.

2.1 System Models

We consider a set of n processes denoted by $\Pi = \{p_1, \ldots, p_n\}$, with $n > 2$. Processes communicate through message passing and do not have access to a shared memory. Each pair of processes in the system is connected by a point-to-point communication channel, except for processes in wireless networks (Chapter 7). All the messages sent on the network are unique and taken from a set $\mathcal{M}$, except for messages sent by a Byzantine process (Part II of the thesis).

Communication model: Processes are connected by a communication network, modeled for each process $p \in \Pi$ by a variable buffer_p, which contains all messages that have been sent to p but not yet received by p [DLS88]. Processes proceed by making *steps*, where a step is either a send or receive step:

- In a *send step*, a single message can be sent to another process in the system, *i.e.*, when process p executes $\mathit{send}(m, p)$ to q, the tuple $\langle m, p \rangle$ is placed in buffer_q.
- In a *receive step*, some messages are received, *i.e.*, when process p executes $\mathit{receive}(M)$, a set $M \subseteq \mathit{buffer}_p$ is removed from buffer_p, and delivered to p. Note that M may be empty.

In each step, some computation can be done.

Configuration: A configuration C of the system consists of the internal state of all processes, together with the content of the buffers. A step brings the system from one configuration C to another configuration C'. In the context of consensus problem, defined later in Section 2.4, the *valence* of a configuration C, denoted by $val(C)$ is the set of possible decision values in configurations reachable from C. If $|val(C)| = 1$, the configuration C is *univalent.* If $|val(C)| = 2$, the configuration C is *bivalent.* A configuration is called *v-valent*, if v is the only possible decision value of the configuration [Sch09].

2.1.1 Synchrony

Synchronous system: In a synchronous system there is (i) a known bound Δ on the transmission delay of messages, and (ii) a known bound Φ on the relative speed of processes:

- *Bound on message delay:* If a message m is sent by process p to process q at time t, then q receives the message no later than time $t + \Delta$.
- *Bound on the relative speed of processes:* If the fastest process takes x time units to perform some computation step, then the slowest process does not take more than $x \cdot \Phi$ time units to perform the same computation step.

Asynchronous system: In an asynchronous system there is no bound on the transmission delay of messages and no bound on the relative speed of processes. The consequence is that in an asynchronous system, it is never possible to know whether a process has crashed or not. Therefore, many problems in distributed computing like consensus, atomic broadcast, etc. are not solvable in this model in the presence of a single crash [FLP85].

Partially synchronous system: A partially synchronous system is an asynchronous system that eventually becomes synchronous. Both processes and communication links can be partially synchronous. Partially synchronous models are more realistic than either completely synchronous or completely asynchronous models, since real systems typically do use some timing information. There are two definitions for partially synchronous communication [DLS88]:

- *Unknown bound:* There is a bound on the transmission delay of messages, and a bound on the relative process speed, but the value of Δ and Φ are unknown (the bounds depend on the run).
- *Known bounds Δ and Φ hold eventually:* There exists a known Δ and Φ with the following property: For every run R, there is an unknown

time, called *Global Stabilization Time* (GST), such that the transmission delay of messages is bounded by Δ, and the relative process speed is bounded by Φ after GST. It is also assumed that channels can lose messages before GST, but are reliable (or quasi-reliable) after GST.

Chandra and Toueg define a weaker model of partial synchrony [CT96]:

- *Unknown bound holds eventually:* There is a bound on the transmission delay of messages, and a bound on the relative speed of processes, but they are unknown and hold only after some unknown time GST.

Consider an algorithm $\mathcal{A}$ and different runs of the algorithm. Let $\mathcal{R}_{sync}$ denote the set of synchronous runs of the algorithm, $\mathcal{R}_{part\_sync}$ denote the set of partially synchronous runs of the algorithm, and finally $\mathcal{R}_{async}$ denote the set of asynchronous runs of the algorithm. Following relation holds between different runs of algorithm $\mathcal{A}$:

$$\mathcal{R}_{async} \supset \mathcal{R}_{part\_sync} \supset \mathcal{R}_{sync}.$$

Asynchronous system augmented with unreliable failure detectors: One way to overcome the impossibility result in asynchronous system, is augmenting it with the notion of *failure detector* [CT96]. Each process p_i has access to a local *failure detector model* (FD_i) that it can query. Each local module FD_i monitors the processes in the system and maintains a list of those that it currently suspects to have crashed. Moreover,

- Each local failure detector can make a mistake by erroneously adding processes to its list of suspects (*i.e.*, it can suspect a process that has not crashed). In other words, the failure detectors are *unreliable.*
- A local failure detector can change its mind by removing a process from its list, if it believes that suspecting a process was a mistake.
- At a given time the failure detector modules at two different processes may have different lists of suspects.

If we do not set any constraint on the output of the failure detectors, the model does not add anything with respect to an asynchronous system, and the FLP impossibility result still holds. The idea is to put requirements on the output of the failure detectors. These requirements are expressed in terms of two abstract properties, a *completeness* property and an *accuracy* property. Completeness defines constrains with respect to crashed processes, while accuracy defines constrains with respect to correct processes. For instance, we have:

- *Strong completeness:* Eventually every process that crashes is permanently suspected by *every* correct process.

- *Eventual strong accuracy:* There is a time after which correct processes are not suspected by *any* correct processes.

A pair *(completeness property, accuracy property)* defines a failure detector. We will only refer to the following failure detector in this thesis (see Chapter 4):

- *Eventual perfect failure detector $\Diamond\mathcal{P}$:* satisfies strong completeness and eventual strong accuracy.

2.1.2 Fault model

In the thesis we consider both *Byzantine* and *benign* (non-Byzantine) faults.

Benign fault: In the benign fault model, processes follow their protocol correctly, but they can crash at any time. In the first and last part of thesis we use the notion of *transmission fault* instead of faulty component, which handles the benign faults, being static or dynamic, permanent or transient, in a unified way.

Transmission fault: Consider process p that is supposed to send a message to process q. With failure detectors, q waits for the message from p unless it suspects p. If p is correct, p may never be suspected. So, to avoid q from being blocked, the solution requires the link between p and q to be reliable (see Section 2.1.3). As a result, the fault model is asymmetric: processes can fail, links never fail. Moreover, if messages are not received, the responsibility is usually put on processes (sender), not on links. This leads to the following question: *is it actually necessary to put the responsibility of the non reception of messages on some component (process or link)?* The answer is classically "yes", as explained in [CBS07], even though finding a "culprit" for the fault is actually not only irrelevant when solving *e.g.* consensus, but is even harmful, since it leads to reasoning about transient faults in terms of permanent faults. Indeed, if the real source for the non reception of a message is some component C (link or process), and C is not permanently faulty, identifying C and labeling it as "faulty" makes no sense: in the future C might behave correctly. The logical consequence is that, when handling transient faults, the right approach is to abstain from the notion of "faulty" components.

Following [SW89], we ignore such culprits by introducing *transmission faults*. When p is supposed to send message m to q, and m is not received, we say that a transmission fault has occurred. A transmission faults does not put the blame on any component, neither on p or q, nor on the link between p and q. This simple approach allows us to handle permanent and transient failures, *i.e.*, crash-stop, crash-recovery and (transient) link faults, uniformly in the context of benign faults.

Byzantine fault: In the Byzantine fault model, processes may exhibit a completely unconstrained behavior. Byzantine faults model a bit flip in the memory of process p that leads p to send a message m' different from m the message it should have normally sent. Byzantine faults also model a process that deliberately tries to deceive the other processes in the system, by sending incorrect messages.

With Byzantine faults, a *faulty* process can crash or behave maliciously. A *correct* process is a non-faulty process. Parameter t is a bound on the number of faulty processes.

In the context of Byzantine faults, the assumption is that the immediate sender of any message can be identified. In other words, when p receives a message from q, it knows that the message was sent by q, and not by some process q' that tries to impersonate q. This is referred as *authenticated channels* in the literature and can be implemented using *symmetric cryptography* in practice. For example, if the channel between p and q is authenticated, then p knows that the messages received on this channel was sent by q. It is also assumed that messages are not corrupted during the transmission. Lamport refers to this fault model as *oral messages* in [LSP82].

Authenticated Byzantine fault: There is a different model in which messages are signed (using digital signatures based on public key cryptography) with an assumed unforgeable signature. Consider process p that receives a message m signed by q (notation $m : q$). Whether p receives this message directly from q or through another process q', p can rely on the fact that the message was sent by q. This model is called *authenticated Byzantine faults* [DLS88]. Lamport refers to this fault model as *written messages* in [LSP82].

2.1.3 Communication channels

With respect to links (channels) the following definitions can be considered:

- *Integrity (no creation, no duplication):* Process q receives a message m from process p at most once, and only if p previously sent m.
- *Reliable link:* A reliable link satisfies integrity and the following property: If p sends message m to q and q is correct, then q eventually receives m.
- *Quasi-reliable link:* A quasi-reliable link satisfies integrity and the following property: If p and q are correct and p sends message m to q, then q eventually receives m.
- *Fair link:* A fair link satisfies integrity and the following property: If p sends infinitely many messages to q and q is correct, then q receives infinitely many messages from p.
- *Lossy link:* A lossy link satisfies only integrity and may lose messages.

We have the following relation between different links:

$$\textit{Reliable link} \Rightarrow \textit{Quasi-reliable link} \Rightarrow \textit{Fair link} \Rightarrow \textit{Lossy link}.$$

Let process q be correct. Assuming reliable links, if p sends message m to process q at time t, and crashes at time $t+1$, then q must eventually receive m. This is not the case with quasi-reliable links, which defines a weaker property. The reliable link definition does not adequately model existing transport layers (*e.g.*, TCP). However, reliable links are useful to prove impossibility results. Two definitions are equivalent if processes do not crash [Sch09].

2.2 Heard-Of Model and Benign Faults

In the context of benign faults, we consider the *Heard-Of* model [CBS09]. The *Heard-Of* model (*HO* for short) is a computational model for distributed systems that combines the advantages of the *Round-by-Round Fault Detectors (RRFD)* model of Gafni [Gaf98] (an extension of the round model introduced by Dwork, Lynch, Stockmeyer [DLS88]) and the *Transmission Fault* model [SW89, SW90], but avoids their drawbacks. HO model is a round-based model in which (1) synchrony degree and failure model are encapsulated in the same high-level abstraction, and (2) the notion of faulty component (process or link) has totally disappeared. As a result, the HO model accounts for transmission faults without specifying by whom nor why such faults occur. The HO model handles benign faults, being static or dynamic, permanent or transient, in a unified way.

HO algorithm: A computation in the HO model evolves in rounds. An algorithm in the HO model consists, for each round r and process $p \in \Pi$, of a sending function S_p^r and a transition function T_p^r. Let s_p denote the current state of process p. For each round r and each p, the sending function $S_p^r(s_p)$ determines a vector of messages to be sent, one message for each process (*null* if there is no message for this process). At the end of a round r, p makes a state transition according to $T_p^r(\vec{\mu}, s_p)$, where $\vec{\mu}$ is the partial vector of messages received in round r. Rounds are communication-closed: a message sent in round r to q and not received by q in round r is lost.

HO predicate: We denote by $HO(p,r)$ the set of processes (including itself) from which p receives a message at round r: $HO(p,r)$ is the *heard of* set of p in round r. If $q \notin HO(p,r)$, then the message sent by q to p in round r was subject to a transmission failure. Communication predicates are expressed over the sets $(HO(p,r))_{p\in\Pi, r>0}$. For example,

$$\exists r_0, \forall p, q \in \Pi : HO(p, r_0) = HO(q, r_0)$$

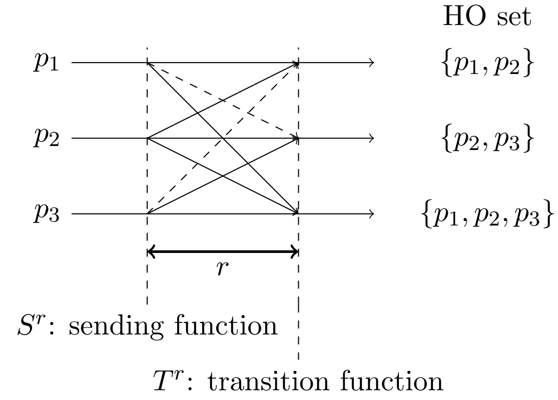


Figure 2.1: Heard-Of model and HO sets.

ensures the existence of some round r_0 in which all processes hear of the same set of processes. In the thesis we refer to this round as a *space uniform* round. Another example is a communication predicate that ensures that in every round r, all processes hear of a majority of processes:

$$\forall r > 0, \forall p \in \Pi : |HO(p, r)| > n/2.$$

Figure 2.1 shows the HO set of processes in round r, assuming that a processes always hears from itself.

HO machine: Let $\mathcal{A} = \langle S_p^r, T_p^r \rangle$ be an HO algorithm (based on the S_p^r sending function and T_p^r transition function) and $\mathcal{P}$ a communication predicate over the HO sets. The tuple $\langle \mathcal{A}, \mathcal{P} \rangle$ specifies an HO machine that can be used to solve a problem.

A coordinated HO machine (CHO) is an extension of an HO machine that includes the notion of coordinator. This allows the specification of coordinator-based algorithms, by giving predicates not only over the HO sets but also over the current coordinator. In a CHO machine, $Coord(p, r)$ denotes p's coordinator at round r. The sending function S_p^r and transmission function T_p^r take the current coordinator as an additional parameter, reflecting the fact that the messages to be sent and the state transitions depend also on the coordinator.

Consensus algorithms, including coordinator-based algorithms, consist of a sequence of one or more rounds that are repeatedly executed. This sequence of one or more rounds is called a *phase*. Typically, the coordinator is changed only at the beginning of a phase. This is the case of all the coordinated algorithms we consider in the thesis; therefore we will use the notation $Coord(p, \phi)$ to refer to the coordinator of process p during all the rounds of phase ϕ.

2.3 Round Model and Byzantine Faults

In the context of Byzantine faults, the classical approach is to distinguish a faulty process from a correct one and put the synchrony assumption only on the correct processes. In [BCBG$^+$07], the authors have extended the HO model for the *value faults*, which affect only the information that is exchanged among processes and do not affect the state of the processes. They obtain results that are not comparable with classical results.

In this thesis, we keep the classical definition of Byzantine faults and use the basic round model [DLS88] on top of the system model – instead of the HO model – in order to obtain the results that match the classical results. In the round model, processing is divided into rounds of message exchange. To be consistent with the rest of the thesis, we use the same notation as in the HO model but we require predicates that hold only for correct processes. We denote the set of correct processes by $\mathcal{C}$ ($\mathcal{C} \subset \Pi$).

The message sent by a correct process in round r is denoted by σ_p^r; messages received by process p in round r are denoted by $\vec{\mu}_p^r$ ($\vec{\mu}_p^r$ is a vector, with one entry per process; $\vec{\mu}_p^r[q] = \perp$ means that p received no message from q).

A round r fulfills *integrity* if any message received from a correct process q in round r was sent in r by q, or no message is received:

$$\mathcal{P}_{Int}(r) \equiv \forall p, q \in \mathcal{C} : \vec{\mu}_p^r[q] \in \{\sigma_q^r, \perp\}$$

Synchronous round: In a partially synchronous system it is possible to ensure the following property: there exists some round *GSR* (*Global Stabilization Round*) such that for all rounds $r \geq GSR$, the message sent in round r by a correct process q to a correct process p is received by p in round r:

$$\mathcal{P}_{Sync}(r) \equiv \forall p, q \in \mathcal{C} :\ \vec{\mu}_p^r[q] = \sigma_q^r$$

We say that such a round is *synchronous*, and assume that

$$\exists GSR \text{ s.t. } \forall r \geq GSR : \mathcal{P}_{Sync}(r)$$

A synchronous round satisfies also integrity.

Consistent round: We then define a *consistent* round as a round in which all correct processes receive the same set of messages:

$$\mathcal{P}_{Cons}(r) \equiv \forall p, q \in \mathcal{C} :\ \vec{\mu}_p^r = \vec{\mu}_q^r$$

WIC round: A round in which $\mathcal{P}_{Sync}$ and $\mathcal{P}_{Cons}$ hold is called a *WIC round.*

$$\mathcal{P}_{WIC}(r) \equiv \forall p, q \in \mathcal{C} :\ \mathcal{P}_{Sync}(r) \wedge \mathcal{P}_{Cons}(r)$$

The notion of WIC (Weak Interactive Consistency) abstraction has been introduced in [MHS09] as following:

$$(\forall r: \ \mathcal{P}_{Int}(r)) \wedge (\exists r_0, \mathcal{P}_{WIC}(r_0))$$

2.4 Agreement Problems

In this thesis, we concentrate on the well-known agreement problem, called *Consensus*. In the first and third part of the thesis we consider the consensus problem in the context of benign faults, while the second part considers Byzantine consensus problem.

2.4.1 Consensus problem

Consensus is defined over the set of processes Π, where each process $p \in \Pi$ has an initial value v_p: all processes must agree on a common value that is the initial value of one of the processes. The problem is formally specified by the following conditions:

- *Validity:* Any decision value is the initial value of some process.
- *Uniform agreement:* No two processes decide differently.
- *Termination:* All processes eventually decide.

In the case of HO model, where there is no notion of faulty process, a process is never exempted from making a decision. The termination property is discussed in [CBS09].

2.4.2 Byzantine consensus

With Byzantine faults the consensus problem cannot be specified in the same way as for benign (non-Byzantine) faults. First, the uniform agreement property does not make sense: we cannot force a Byzantine process to decide on some value v, *i.e.*, the Byzantine process can always decide on $v' \neq v$. Second, the validity property needs also to be adapted: we must prevent that the decision is always on the initial value of Byzantine processes. Indeed, in this case, correct processes would not be able to impose a decision. At least two validity properties have been considered in the literature [DLS88]:

- *Agreement:* Two correct processes cannot decide differently.
- *Termination:* All correct processes eventually decide.
- *Strong validity:* If all correct processes have the same initial value v, then a correct process can only decide v.
- *Weak validity:* If all processes are correct and if a (correct) process decides v, then v is the initial value of some process.

Weak validity allows correct processes to decide on the initial value of a Byzantine process. With strong validity, however, this is only possible if not all correct processes have the same initial value. Another weaker validity property, called *external validity* can be found in the literature [CKPS01] for multi-valued Byzantine agreement. In the thesis, we consider only strong validity and give algorithms that satisfy the strong validity property of Byzantine consensus.

2.4.3 Interactive consistency

Byzantine consensus was first identified by Pease, Shostak and Lamport [PSL80], formalized as the *Interactive Consistency* problem and solved in a synchronous system. An algorithm achieves interactive consistency if it allows the nonfaulty processes to come to a consistent view of the initial values of all the processes, including the faulty ones. In other words, each correct process computes a vector of values, with one element for each of the n processes, such that:

- The correct processes compute exactly the same vector.
- The element of the vector corresponding to a given correct process is the initial value of that process.
- The element of the vector corresponding to a given faulty process is either some common value v or $\perp$.

2.5 Broadcast Protocols

Following broadcast protocols exist in the literature [Sch09]:

2.5.1 Consistent Broadcast

Consistent broadcast [BT83, Tou84] is defined in the context of Byzantine faults. Consider a process p that must broadcast some message: nothing prevents p from sending a different message to the different processes. Consistent broadcast is a broadcast primitive that ensures that the delivered message is the same for all correct processes. However, it does not guarantee that *every* correct process delivers a message. It is defined by two primitives *cbcast* and *cdeliver*:

- *Validity:* If a correct process $cbcast(m)$, then all correct processes eventually $cdeliver(m)$.
- *Consistency:* If a correct process $cdeliver(m)$ and another correct process $cdeliver(m')$, then $m = m'$.
- *Integrity:* For any message m, every correct process $cdeliver(m)$ at most once, and only if m was previously cbcast.

These properties are only safety properties.

2.5.2 Reliable Broadcast

Reliable broadcast [Bra84, Bra87, ST87b] is consistent broadcast that additionally ensures agreement on the delivery of the message in the sense that either all correct processes deliver some message or none delivers any message. It is defined by two primitives *rbcast* and *rdeliver*:

- *Agreement:* If a correct process rdelivers a message m, then all correct processes eventually rdeliver m.

2.5.3 Atomic Broadcast

Atomic broadcast [HT93] is a reliable broadcast that additionally adds an order on the delivery of the messages. It is defined by two primitives *abcast* and *adeliver*:

- *Total order:* If some correct process adelivers m before m', then every correct process adelivers m' only after it has adelivered m.

There are mainly two kinds of atomic broadcast protocols [DSU04]:

1. Protocols that use the consensus module to order the messages, as shown in [CT96] for the benign faults;
2. Protocols that are sequencer based and do not rely on consensus in a modular way, like Paxos [Lam98] and PBFT [CL02].

Both consistent and reliable broadcasts are defined for a single message, while atomic broadcast is defined for a sequence of messages. With benign faults, the uniform version of reliable and atomic broadcasts are well-defined.

2.5.4 Terminating Reliable Broadcast

Terminating reliable broadcast, also called Byzantine Generals' Problem [LSP82], is a reliable broadcast with an additional *termination* property and a modified integrity property. It is defined by two primitives *trbcast* and *trdeliver*:

- *Integrity:* If a correct process trdelivers m, then either $m = \perp$ or m was trbcast.
- *Termination:* Every correct process eventually trdelivers exactly one message.

Remark I: The interactive consistency problem is a generalization of the terminating reliable broadcast problem. Therefore, it can be used to solve consensus.

Remark II: Since reliable broadcast can be solved in an asynchronous system, it is easier than the consensus problem. However, from [CT96] we know that the terminating reliable broadcast cannot be solved even in an eventual synchronous system; therefore it is harder than the consensus problem.

Remark III: Contrary to the consistent and reliable broadcast, the WIC abstraction [MHS09] is defined in the round model, is an all-to-all primitive, and is required to solve consensus. In asynchronous rounds, it satisfies the properties weaker than the consistent broadcast (two correct may receive different messages from a Byzantine), while in synchronous rounds, it satisfies the properties of the reliable broadcast.

2.6 State Machine Replication

State machine replication is a general approach to implement a fault-tolerant service by replicating servers and coordinating client interactions with server replicas (client requests are totally ordered). The model assumes that each server is a deterministic finite state machine. Lamport was the first to propose state machine replication, in 1984 in his seminal paper [Lam84]. Paxos algorithm of Lamport [Lam98, Lam01] is the most referenced fault-tolerant algorithm in the literature and the most used in practice. The algorithm is leader-based, tolerates a minority of crashes, and does not require reliable links. The algorithm is originally expressed in terms of “proposers”, “acceptors”, and “learners” and the description of the algorithm is rather complex. However its consensus core is not that complicated as recently presented in the HO model by Charron-Bost and Schiper [CBS09], called *LastVoting*. An implementation of the *LastVoting* algorithm is discussed in the thesis (see Chapter 3 and 7).

In the context of Byzantine faults, the most widely referenced Byzantine fault tolerant protocol in the literature is the PBFT protocol of Castro and Liskov [CL02]. PBFT can actually be viewed as a Byzantine-fault-tolerant version of the Paxos protocol, or of viewstamped replication [OL88]. PBFT uses a sequencer-based approach and is rather complex. The consensus core of the algorithm can be better understood using the round model and the WIC abstraction presented in [MHS09], called *CL*. The *CL* consensus algorithm can use either a leader-based implementation of WIC (as shown in [MHS09] and Chapter 6) or a decentralized implementation of WIC (see Chapter 5). An implementation of the *CL* algorithm is discussed in the thesis (see Chapter 6).

There are a huge amount of system papers that tried to improve PBFT in some way. One direction is the number of replicas without impacting resiliency, addressed first in [YMV$^+$03] and continued in [WSVS08]. Another direction is improving the best case performance, started by Zyzzyva

[KAD$^+$07] and continued in [SS08, SBD$^+$10]. More modular BFT protocols are proposed in [CKL$^+$09, GKQV10]. However, any improvement to the original BFT protocol may impact its main property that is Byzantine fault tolerance. The problem was addresses recently in several papers [ACKL08, CWA$^+$09, VCBL09]. One way to understand better the PBFT protocol together with its advantages and disadvantages is extracting its consensus core and analyzing it separately as done in this thesis.

There are other approaches to implement a Byzantine fault-tolerant state machine replication in the literature (see [CBPE10]): (1) quorum systems [MR98, Baz00, AEMGG$^+$05, CML$^+$06, DBFC07, GV07]; (2) randomization [CKS05, CP02]; (3) hybrid protocols [CML$^+$06]. Quorum systems and randomization are not considered in the thesis.

Part I

Benign Faults

Chapter 3

Quantitative Analysis of Consensus Algorithms

In Section 2.2 we have introduced the HO model and communication predicates in the context of benign faults. This chapter discusses several implementations of communication predicates in a system that alternates between good periods and bad periods. In this context, we quantify the required length of a good period to solve one or more instances of consensus. With our results, we can observe several interesting issues, for example that the number of rounds is not in general a good prediction for the time to solve consensus.

Publication: F. Borran, M. Hutle, N. Santos and A. Schiper. Quantitative Analysis of Consensus Algorithms. Technical Report (submitted to *IEEE Transactions on Dependable and Secure Computing (TDSC)*), EPFL, 2010. The chapter is a joint work with Nuno Santos.

3.1 Introduction

Consensus is one of the key problems in fault tolerant distributed computing. It has been shown that consensus is solvable in a partially synchronous system with a majority of correct processes [DLS88]. Consensus is also solvable in an asynchronous system "augmented" with failure detectors [CT96]. Over the years failure detectors have become very popular. The model is today widely accepted, and has become the most common model used for expressing consensus algorithms.

However, the fact that consensus is solvable with failure detectors or in a partially synchronous system does not close the problem: ensuring the termination property does not address the performance issue. In other words, a quantitative analytical comparison of consensus algorithms is needed.

The problem of analytical quantitative evaluation of consensus algorithms was initially addressed in [DLS88], in the context of a partially synchronous system. The authors compute upper bounds for the time needed after GST (Global Stabilization Time) for all correct processes to decide, for algorithms proposed in the paper. The upper bounds are computed for different variants of a partially synchronous system. We could not compare our results with the results of [DLS88], since we consider different consensus algorithms (the algorithms in [DLS88] have been ignored since). After this early work, analytical performance evaluation of consensus algorithms has not received much attention for a while. This is probably due to the advent of failure detectors, which led to consider an asynchronous underlying system, and to ignore timing analysis. One of the first paper to reinitiate analytical performance study of consensus algorithms in non-synchronous systems is [Sch97]. The paper considers failure detectors, and uses as metric the minimum number of communication steps for deciding in a "nice" run (run with no crashes, no false suspicions). Later, [DGK07], [KS06], and [AGGT08] study the performance of consensus algorithms expressed in a round-based computational model. The performance metric is the number of rounds needed for processes to decide once the system has become synchronous. However, as pointed out in [KS06], the efficiency expressed in terms of number of rounds, does not predict the time it takes to decide after the system stabilizes. This is recognized in [KS06] as an interesting subject for further studies. Such a timing analysis is done in [DGL05] for a modified version of Paxos. The authors show that, with their modified Paxos algorithm, consensus can be solved in $O(\delta)$ after the system stabilizes (actually 17δ), where δ is the upper bound on message delivery time after stability is reached (δ includes the time needed to process a message after reception). Timing analysis is also done in [PLL97] for Paxos, but only for an execution started during a good period, and leader election outside of the Paxos algorithm (it uses a failure detector implementation in which every process sends periodically messages to all). The Paxos algorithm, when expressed in the failure detector model with reliable links, requires the Ω leader oracle. [ADGFT03] shows that such a leader election service can be implemented in a *communication-efficient* way, which means that there is a time after which only one process sends message. LastVoting (the algorithm is given in Section 3.2.1.A) with synchronization by a coordinator achieves the same efficiency: in a good period only n messages are required in phase 4ϕ. However, our solution has an advantage: it only uses the "natural" message of LastVoting, while a leader election service needs to send its own messages in addition to the consensus algorithm messages. In other words, our solution is even more "communication-efficient" than a solution based on an external communication-efficient leader election service.

The goal of the chapter is to go beyond the above mentioned work, in order to propose a more general timing analysis of some consensus algorithms. As in [DLS88], the analysis is done for algorithms expressed in a round-based model built on top of a partially synchronous system. Round-based models allow us to handle permanent and transient faults in a uniform way [CBS09]. Further, such a modular approach decouples timing analysis from irrelevant details of consensus algorithms, which allows us (i) to reuse a given timing analysis for different algorithms, and (ii) to compare various round implementations for the same round-based consensus algorithm. This points to the second contribution of this chapter: in addition to the timing analysis, the chapter proposes (and analyzes) new algorithms for implementing rounds. For the purpose of the timing analysis, the chapter considers a system that alternates between good and bad periods. The notion of good and bad period appears in [CF99], but such a system assumption is not used for a quantitative analysis of consensus algorithms. During a good period, processes do not crash, the system is synchronous (with a known bound Φ on the relative process speeds and a known bound Δ on the transmission delay of messages), and clocks have a bounded drift. During a bad period, processes may crash and recover, messages may be lost and no synchrony assumptions need to hold.

Contribution: Based on Δ, Φ, n (the number of processes), and the accuracy of the clock, the chapter computes the length of the good period for two consensus algorithms (and some of its variants), and different implementations of rounds. Two extreme cases are highlighted: "short" and "long" good periods. The chapter shows that the relative performance (*i.e.*, the number of consensus instances that can be solved in a given good period) of the consensus algorithms analyzed is not the same in the two cases. The explanation is the following: at the beginning of a good period, processes need to synchronize to the same round, and this takes some time; only after synchronization rounds contribute to solve consensus. If a good period is long, the cost of synchronization is amortized among all instances of consensus, and can thus be ignored. This is not the case if the good period is short. The chapter shows that while the considered algorithms show almost identical performance when the good periods are long, this is not the case for short good periods. The chapter shows also that the usual performance metric, namely the number of rounds needed for solving consensus, does not predict our results.

Further, our results allow us to quantify the influence of the clock precision on the length of the good period. It can be seen that a large clock skew, as it is the case when using step counting, has only limited influence for algorithms that resynchronize in every round, but can become unacceptable for algorithms that resynchronize less often.

Roadmap: The structure of the chapter is the following: In Section 3.2 we introduce the consensus algorithms using the HO round model, together with the communication predicates, which form the basis for our analysis. Implementation of communication predicates is the topic of the subsequent sections. After defining our system model for our implementations in Section 3.3, we give an abstract algorithm that may serve as a generic implementation for many predicates, and which is used by all our implementations. After that, in Sections 3.5 to 3.7 we describe how our predicates can be implemented using different strategies. Different strategies lead to different length of the good period that is necessary to guarantee the predicate, and to different number of messages. We finally analyze our results in Section 3.8 and conclude the chapter in Section 3.9.

3.2 Background

3.2.1 Consensus algorithms analyzed

We analyze three consensus algorithms in this chapter that are safe by design (*i.e.*, they never violate the validity or agreement properties of consensus), but require some predicate to ensure liveness. The fact that these algorithms do not require any predicate for safety implies that they are tolerant to transient faults. The safe algorithms are also referred as *indulgent* algorithms [Gue00] in the context of failure detectors; *i.e.*, algorithms that tolerate unreliable failure detection.

3.2.1.A Variant of Paxos: LastVoting in four rounds (LV-4)

The first consensus algorithm we consider is a variant of Paxos [Lam98, Lam01], called *LastVoting* [CBS09], see Algorithm 3.1. It is a variant of Paxos in the sense that the algorithm is expressed here in a round model, which is not the way Paxos has been expressed [Lam98]. It is also close to the Chandra-Toueg $\Diamond\mathcal{S}$ [CT96] consensus algorithm. *LastVoting* is coordinator-based, and each phase of the algorithm consists of four rounds $4\phi - 3$ to 4ϕ (ϕ denotes the current phase). $Coord(p, \phi)$ denotes the coordinator of p in phase ϕ.

We describe briefly how the LV-4 algorithm works: Each process p has a timestamp ts_p attached to its proposal x_p. (1) In the first round of every phase, each process sends its proposal and timestamp to its coordinator (line 6). If the coordinator receives proposals from a majority of processes, it sets its *vote* to the last proposal with the highest timestamp (line 13). (2) In the second round, the coordinator sends its *vote* to all (line 18). Every process that receives coordinator's vote (line 20), changes its proposal and updates its timestamp. (3) These processes send an *ack* message to the coordinator in the third round (line 26). If the coordinator receives a majority of *ack*s (line 28), it can decide on its *vote*. (4) The coordinator sends its *vote* (the decision) to all processes in the last round (line 33), and each process that receives the coordinator's *vote* decides (line 36). Note that process p is not blocked in a round: if the conditions at lines 11, 20, 28 and 35 are false in some round r for process p, then process p skips the corresponding T_p^r part.

The correctness proof of the LV-4 algorithm can be found in [CBS09]. The algorithm is always safe even if there are several coordinators per phase. Note that if two coordinators coexist in phase ϕ, because of line 6, the condition of line 11 can be true for at most one coordinator, *i.e.*, at most one coordinator can send a proposal in phase ϕ at line 18. The termination property of the algorithm is guaranteed by the existence of a phase ϕ such that following predicate holds:

$$\begin{aligned}\mathcal{P}_{lv4}(\phi) :: \exists \Pi_0 \subseteq \Pi \ s.t. \ |\Pi_0| > n/2, \exists c \in \Pi, \forall p \in \Pi_0 :& \\ \mathit{Coord}(p,\phi) = c \ \wedge& \\ |\mathit{HO}(c, 4\phi - 3)| > n/2 \ \wedge \ c \in \mathit{HO}(p, 4\phi - 2) \ \wedge& \\ \Pi_0 \subseteq \mathit{HO}(c, 4\phi - 1) \ \wedge \ c \in \mathit{HO}(p, 4\phi)&\end{aligned}$$

which ensures, loosely speaking, agreement on the coordinator c during one phase ϕ, and communication between c and a majority of processes during phase ϕ.

3.2.1.B Variant of Paxos: LastVoting in three rounds (LV-3)

LastVoting in three rounds is a well known variant of Paxos in which the last two rounds of a phase are aggregated in a single round [CBS06], as shown by Algorithm 3.2: in round 3ϕ all processes send their *ack* message directly to all other processes, instead of via the coordinator. LV-3 terminates in a phase ϕ satisfying the following predicate:

$$\begin{aligned}\mathcal{P}_{lv3}(\phi) :: \exists \Pi_0 \subseteq \Pi \ s.t. \ |\Pi_0| > n/2, \exists c \in \Pi, \forall p \in \Pi_0 :& \\ \mathit{Coord}(p,\phi) = c \ \wedge \ |\mathit{HO}(c, 3\phi - 2)| > n/2 \ \wedge& \\ c \in \mathit{HO}(p, 3\phi - 1) \ \wedge \ \Pi_0 \subseteq \mathit{HO}(p, 3\phi).&\end{aligned}$$

3.2.1.C OneThirdRule (OTR)

Contrary to LV-4 and LV-3, which are both coordinator-based algorithms, Algorithm 3.3 does not use a coordinator (referred as a decentralized algorithm in this thesis). This algorithm, called *OneThirdRule* (or simply

Algorithm 3.1 LV-4: *LastVoting in four rounds* [CBS09]
(code of process p)

```
Initialization:
  $x_p := v_p \in V$                                   /* $v_p$ is the initial value of $p$ */
  $vote_p \in V \cup \{?\}$, initially ?
  $commit_p$ a Boolean, initially false
  $ready_p$ a Boolean, initially false
  $ts_p \in \mathbb{N}$, initially 0

Round $r = 4\phi - 3$:
  $S_p^r$:
    send $\langle x_p, ts_p \rangle$ to $Coord(p, \phi)$
  $T_p^r$:
    if $p = Coord(p, \phi)$ and number of $\langle \nu, \theta \rangle$ received $> n/2$ then
      let $\overline{\theta}$ be the largest $\theta$ from $\langle -, \theta \rangle$ received
      $vote_p :=$ one $\overline{x}$ such that $\langle \overline{x}, \overline{\theta} \rangle$ is received
      $commit_p :=$ true

Round $r = 4\phi - 2$:
  $S_p^r$:
    if $p = Coord(p, \phi)$ and $commit_p$ then
      send $\langle vote_p \rangle$ to all processes
  $T_p^r$:
    if received $\langle v \rangle$ from $Coord(p, \phi)$ then
      $x_p := v$
      $ts_p := \phi$

Round $r = 4\phi - 1$:
  $S_p^r$:
    if $ts_p = \phi$ then
      send $\langle ack \rangle$ to $Coord(p, \phi)$
  $T_p^r$:
    if $p = Coord(p, \phi)$ and number of $\langle ack \rangle$ received $> n/2$ then
      $ready_p :=$ true

Round $r = 4\phi$:
  $S_p^r$:
    if $p = Coord(p, \phi)$ and $ready_p$ then
      send $\langle vote_p \rangle$ to all processes
  $T_p^r$:
    if received $\langle v \rangle$ from $Coord(p, \phi)$ then
      DECIDE($v$)
    $commit_p :=$ false
    $ready_p :=$ false
```

OTR), appears in [CBS09]. It has similarities with a fast round of the Fast Paxos algorithm [Lam05]. Every round of OTR has the same sending and transition function. Decision can be reached in one round if all initial values are identical, otherwise decision can be reached in two rounds. For liveness, two distinct rounds (not necessarily consecutive) that satisfy the following predicate are needed:

Algorithm 3.2 LV-3: *LastVoting in three rounds* [CBS06] (code of process p)

```
Initialization:
  $x_p := v_p \in V$                                  /* $v_p$ is the initial value of $p$ */
  $vote_p \in V \cup \{?\}$, initially ?
  $commit_p$ a Boolean, initially false
  $ts_p \in \mathbb{N}$, initially 0

Round $3\phi - 2$: identical to round $4\phi - 3$ of Algorithm 3.1.

Round $3\phi - 1$: identical to round $4\phi - 2$ of Algorithm 3.1.

Round $r = 3\phi$:
  $S_p^r$:
    if $ts_p = \phi$ then
      send $\langle ack, x_p \rangle$ to all processes

  $T_p^r$:
    if $\exists v$ such that number of $\langle ack, v \rangle$ received $> n/2$ then
      DECIDE($v$)
    $commit_p$ := false
```

Algorithm 3.3 The *OneThirdRule* algorithm [CBS09] (code of process p)

```
Initialization:
  $x_p := v_p$                                        /* $v_p$ is the initial value of $p$ */

Round $r$:
  $S_p^r$:
    send $\langle x_p \rangle$ to all processes

  $T_p^r$:
    if $|HO(p,r)| > 2n/3$ then
      $x_p$ := the smallest most often received value
    if more than $2n/3$ values received are equal to $v$ then
      DECIDE($v$)
```

$$\mathcal{P}_{su}(r) :: \exists \Pi_0 \subseteq \Pi \ s.t. \ |\Pi_0| > 2n/3, \forall p \in \Pi_0 : \\ HO(p,r) = \Pi_0.$$

We call such a round *space uniform round with cardinality* $2n/3$, and use the term *space uniform round* when the cardinality is clear from the context. The predicate $\mathcal{P}_{otr}$ will denote the existence of two distinct rounds (not necessarily consecutive) that satisfy $\mathcal{P}_{su}(\,)$.

Remark The algorithms and predicates given in this section ensure a decision of a majority of processes (two-thirds majority in case of OTR). However, with a small modification, all processes that are eventually reachable

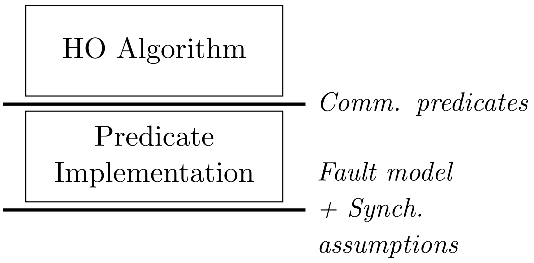


Figure 3.1: The two layers in the HO model when implementing predicates.

will decide: since our agreement property is a uniform property (there are no "faulty" processes that are exempted from agreement), processes — once they have decided — can simply communicate their decision to all other processes. Once this communication is successful, also these processes will decide.

3.2.2 Relation of predicates

Predicates can be related in terms of *translations* [CBS09]. Without explaining this concept in detail, it is easy to see that a sequence of space-uniform rounds "almost" ensures $\mathcal{P}_{lv4}$ (for LV-4) and $\mathcal{P}_{lv3}$ (for LV-3): only the election of the coordinator is missing. However, we can observe that a space-uniform round r allows the election of a unique coordinator for round $r+1$: the coordinator can be determined through a deterministic function on the HO sets of round r. Thus space uniformity of round $4\phi_0$ and of the four rounds of phase ϕ_0+1 allows us to ensure $\mathcal{P}_{lv4}$. Space uniformity of such five consecutive rounds requires eight space-uniform rounds in the worst case (the first space-uniform round does not necessarily correspond to the last round of a phase). Similarly, space uniformity of round $3\phi_0$ and of the three rounds of phase ϕ_0+1 allows us to ensure $\mathcal{P}_{lv3}$. Space uniformity of such four consecutive rounds requires six space-uniform rounds in the worst case.

More generally, $2y$ consecutive space-uniform rounds ensure y instances of $\mathcal{P}_{otr}$, $3y+3$ consecutive space-uniform rounds ensure y instances of $\mathcal{P}_{lv3}$, and $4y+4$ consecutive space-uniform rounds ensure y instances of $\mathcal{P}_{lv4}$. We will use these results in Section 3.5.

3.2.3 Implementation of predicates

In this chapter we are interested in the question of how an HO machine $\langle\mathcal{A},\mathcal{P}\rangle$, where $\mathcal{P}$ is some predicate, can be implemented in a "classical" message-passing model. Fig. 3.1 illustrates how these parts work together in a system. The top layer, the HO Algorithm $\mathcal{A}$, is defined solely in terms of the sending function S_p^r and transition function T_p^r, and assumes some communication predicate $\mathcal{P}$. The communication predicate $\mathcal{P}$ is implemented by the *Predicate Implementation* layer, which builds on top of the system model. These two layers are independent, apart from the interface defined by the communication predicate. This enforces a clear separation between the high-level computational model of the HO Algorithm and the low-level system model and allows each layer to be developed independently. In the rest of this chapter, we give implementations for the communication predicates specified above.

Given an implementation for some predicate $\mathcal{P}$, we are looking for the *length of a good-period*, *i.e.*, the duration our system has to be "good" at some arbitrary point in time in order to ensure the predicate. We specify this in more detail in Section 3.3.

Note that for coordinator-based algorithms, the predicate implementation layer is also responsible for electing the coordinator. This is in contrast to failure detector based solutions, in which the failure detectors (or the leader election oracle) are provided by some external service. Such a service typically uses heartbeat messages. No such external service is used here. The difficulty is, during a good period, to elect a common coordinator, to resynchronize the processes and exchange the necessary messages to ensure the predicate, within a time as short as possible, and to use as few messages as possible.

3.3 System Model

We describe now the system model for the implementation of the predicate layer. We consider a similar model as in [DLS88], *i.e.*, a partially synchronous system (with known bounds that hold eventually) defined in Section 2.1.1, with some modification to reflect good periods of bounded length. Further, we use clocks instead of a bound on the maximum speed of processes, a more general approach, as we explain later.

Processes proceed by making *steps*. Each step is a send or receive step as described in Section 2.1. In each step, some computation can be done, and the local clock $C_p(t)$ of process p at real time t can be read. We assume that local clocks are monotonically non-decreasing at any time. Real time and the local clock take values from $\mathbb{R}$.

Definition 3.1 (Δ-timely message). *A message m sent at time t by some process to a process p is called* Δ-timely*, if it is received at the latest by the first receive step of p at or after time $t + \Delta$.*

Process synchrony is ensured by making steps at a minimum rate. Note that in contrast to [DLS88], there is no restriction on the maximum speed of processes, since we will use clocks instead of step counting in order to measure time.

Definition 3.2 (Φ-synchronous process in interval I)**.** *The bound Φ is said to hold for a process p in some time interval I, if in any sub-interval of length Φ, p takes at least one step.*

Definition 3.3 (local (α, β) bounded-drift clock in interval I)**.** *A local clock $C_p(t)$ has a bounded-drift in a time interval I, if there are a priori known constants α and β with $0 < \alpha \leq \beta$, so that for any two times t_1, $t_2 \in I$ s.t. $0 < t_1 < t_2$:*

$$\frac{C_p(t_2) - C_p(t_1)}{t_2 - t_1} \in [\alpha, \beta]. \tag{3.1}$$

Note that our clock definition is very general, since it includes other assumptions like the classical bounded-drift clocks ($\alpha = 1 - \rho$, $\beta = 1 + \rho$), whereas the values $\alpha = 1/\Phi$, $\beta = 1$ are obtained asymptotically if step-counting is used for measuring time (this would require a lower bound on the duration of a step, of course).

Definition 3.4 (good period)**.** *Let $\Pi_0 \subseteq \Pi$ be a set of processes. An interval I is a* good period *for Π_0, if there are a priori known bounds $\Phi, \Delta \in \mathbb{N}$, and $\alpha, \beta \in \mathbb{R}$, with $\Phi > 0$, $0 < \alpha \leq \beta$, such that (i) in I all processes in Π_0 are Φ-synchronous and have a local (α, β)-bounded-drift clock in I, (ii) no process that is not in Π_0 makes a step, (iii) all links between processes in Π_0 are Δ-timely, and (iv) no messages from processes not in Π_0 are received by a process in Π_0.*

A *k-good period* is a good period for some arbitrary Π_0 with $|\Pi_0| \geq k$. In the sequel, when k is clear from the context we will use only the term *good period.*

Note that we do not specify why processes outside Π_0 do not make steps; they might have crashed, or be just temporarily unavailable. Therefore the notion of *correct* or *faulty* process is not suitable in our context; however, with respect to some Π_0-good period, we say a process is *up* in this good period iff it is in Π_0, else it is *down.*

The *length of a good period* refers to the duration of a good period that is *sufficient* to satisfy some communication predicate. We will also compute the *message complexity* of our implementations. This is the number of messages exchanged by processes in good periods to satisfy the communication predicate.

When setting process timeouts, we will use the following result:

Lemma 3.1. *In a good period, a time interval of length $\tau_C = \beta\tau_L$, measured by some process p, corresponds to a real time interval of length greater or equal to τ_L, and smaller or equal to $\tau_U = \frac{\beta}{\alpha}\tau_L$.*

Proof. Assume w.l.o.g. that the timeout $\beta\tau_L$ starts at time 0 and expires at some real time t, *i.e.*, $C_p(t) = \beta\tau_L$. Because of (3.1) we have $C_p(t)/t \leq \beta$ and $C_p(t)/t \geq \alpha$. Thus $t \geq \frac{C_p(t)}{\beta} = \tau_L$ and $t \leq \frac{C_p(t)}{\alpha} = \frac{\beta}{\alpha}\tau_L$. □

We will keep the notation of τ_L, τ_C, and τ_U consistent within the chapter to denote these different kind of durations.

3.4 The Generic Protocol

We give in this section a generic algorithm for the predicate layer, an algorithm that is parametrized by four abstract functions. The instantiation of these functions will allow us to devise three different algorithms for the predicate layer that differ mainly by the message pattern and the way the coordinator is elected. The first method is called *Full Synchronization* (Section 3.5); it ensures space-uniform rounds, therefore allowing the implementation of all predicates considered in this chapter. *Phase Synchronization* (Section 3.6) is an optimized implementation for $\mathcal{P}_{lv3}$, where round synchronization takes place only once per phase. Finally, *Synchronization by a Coordinator* (Section 3.7), where synchronization uses only messages from coordinator process(es), is specialized for $\mathcal{P}_{lv4}$. We only present the main idea of the last method and the results without going into the details. The full description is given in the technical report [BHSS09].

The generic Algorithm 3.4 follows the following pattern. One iteration of the **while** loop (line 6) corresponds to one round: the sending function is called at line 9 and the transition function is called at line 20. Messages are sent at line 14: the abstract function *Dest* (line 10) specifies the set of processes a message is sent to in the current round. Message reception occurs at line 17. The receive statement is executed repeatedly until *NextRound* returns *true* (line 16). This typically happens when a timer has expired or when a message from some higher round is received. Note also that some rounds may be totally skipped (no message sent, no message received): this happens whenever the function *SkipRound* (line 8) returns *true*, which typically occurs if process p in round r_p receives a message from some round $r' > r_p$. In this case, p skips all rounds from r_p to $r' - 1$. Finally, function *ElectCoord* specifies how a coordinator for each round is determined.

Crash-recovery Model

In order to cope with recoveries after crashes, the variables r_p and s_p of Algorithm 3.4 are stored on stable storage. In case of a recovery, the algorithm starts at line 16, with Rcv_p and t_p reinitialized to $\emptyset$ and 0, respectively. We could express this formally as a variant of Algorithm 3.4, but abstain from this for sake of simplicity. Note that the behavior in case of a sequence

Algorithm 3.4 Generic algorithm of the predicate layer (code of process p)

```
$Rcv_p \leftarrow \emptyset$                                   /* set of messages received */
$r_p \leftarrow 1$                                             /* round number */
$s_p \leftarrow init_p$                                        /* state of the process $p$ */
$coord_p \leftarrow \bot$                                      /* coordinator of process $p$ */
$t_p \leftarrow C_p()$                                         /* timer */
while true do
  $coord_p \leftarrow ElectCoord(p, r_p, C_p() - t_p, coord_p, Rcv_p)$
  if $\neg SkipRound(p, r_p, C_p() - t_p, coord_p, Rcv_p)$ then
    $msgs \leftarrow S_p^{r_p}(s_p, coord_p)$
    for all $q \in Dest(p, r_p, C_p() - t_p, coord_p, Rcv_p)$ do
      if $p = q$ then
        $Rcv_p \leftarrow Rcv_p \cup \{\langle msgs[p], p, r_p \rangle\}$     /* local delivery */
      else
        send$(msgs[q], r_p, p)$ to $q$
    $t_p \leftarrow C_p()$
    while $\neg NextRound(p, r_p, C_p() - t_p, coord_p, Rcv_p)$ do
      receive$(S)$
      for all messages $\langle x, r, q \rangle \in S$ do
        $Rcv_p \leftarrow Rcv_p \cup \{\langle x, q, r \rangle\}$
  $s_p \leftarrow T_p^{r_p}(\{\langle x, q \rangle \mid \langle x, q, r_p \rangle \in Rcv_p\}, s_p, coord_p)$
  $r_p \leftarrow r_p + 1$
```

crash/recovery is completely transparent: although some messages might get lost, no message for a round is sent more than once, and every transition function is called exactly once for each round number.

Reading variables from stable storage is inefficient. The implementation can be made more efficient by keeping a copy of the variables in main memory: a read operation reads the *in memory* copy, a write operation updates the *in memory* and the *stable storage* copies. Upon recovery, the in memory copy is reset with the value of the stable copy.

3.5 Full Synchronization

Our first implementation is given as Parametrization 3.1 for the generic algorithm. The implementation ensures space uniform rounds in a good period. As explained in Section 3.2.2, this is sufficient to implement $\mathcal{P}_{lv3}$ and $\mathcal{P}_{lv4}$.

With Parametrization 3.1, every process sends a message to every other process in all rounds (see function *Dest*). While sending to all in all rounds seems natural for $\mathcal{P}_{otr}$, where every process has to hear from every alive process, this induces some overhead for $\mathcal{P}_{lv3}$ and $\mathcal{P}_{lv4}$, since these predicates only require one-to-all or all-to-one patterns on some of their rounds. This is shown in the following picture for predicate $\mathcal{P}_{lv3}$, where the full lines represent messages required by the predicate and dotted lines represent the

Parametrization 3.1 A generic parametrization using full synchronization

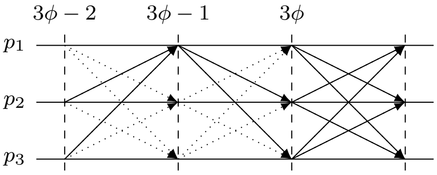
$$\mathit{NextRound}(p, r, \tau, \mathit{coord}, \mathit{Rcv}) := \bigvee \begin{cases} \exists \langle -, -, r' \rangle \in \mathit{Rcv} : \ r' > r \\ \tau \geq \tau_C \end{cases}$$

$$\mathit{SkipRound}(p, r, \tau, \mathit{coord}, \mathit{Rcv}) := \exists \langle -, -, r' \rangle \in \mathit{Rcv} : \ r' > r$$

$$\mathit{Dest}(p, r, \tau, \mathit{coord}, \mathit{Rcv}) := \Pi$$

$$\mathit{ElectCoord}(p, r, \tau, \mathit{coord}, \mathit{Rcv}) := \begin{cases} \min(\Pi) & : \ r = 1 \\ \min(\mathit{ho}(r-1)) & : \ \mathit{ho}(r-1) \neq \emptyset \\ \mathit{coord} & : \ \text{else} \end{cases}$$

additional messages sent by Parametrization 3.1:



In the next sections, we provide implementations for $\mathcal{P}_{lv3}$ and $\mathcal{P}_{lv4}$ with lower message complexity.

3.5.1 Outline of full synchronization

As shown by function *NextRound* in Parametrization 3.1, there are two ways for a process p to leave round r: (i) by receiving a message from a higher round $r' > r$, or (ii) by expiration of a timeout ($\tau \geq \tau_C$). In case (i), the process goes directly to round r': the function *SkipRound* and the first condition of *NextRound* play together to achieve this. In case (ii) the timeout τ_C is chosen to ensure space uniformity, *i.e.*, $\mathcal{P}_{su}(\,)$, in a good period (see Lemma 3.2 below). As shown by the function *ElectCoord*, the coordinator for some round r is the smallest process (min) in the HO set of round $r - 1$ (whenever this HO set is non empty). The definition of the macro $\mathit{ho}(r)$ is the following:

$$\mathit{ho}(r) := \{q \mid \langle -, q, r \rangle \in \mathit{Rcv}\}.$$

We will also use this notation in the following sections. *ElectCoord* ensures a unique coordinator in good periods where rounds are space uniform. For non-coordinated predicates, like $\mathcal{P}_{su}(\,)$, no coordinator is needed and the function *ElectCoord* can be ignored.

3.5.2 Timeout τ_C

We first assume that a good period, which starts at some time t_g, holds forever, and show that the timeout $\tau_C = [2\Delta + (2n-1)\Phi]\beta$ ensures $\mathcal{P}_{su}(\,)$

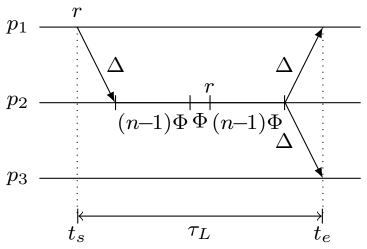


Figure 3.2: Full synchronization: timeout - Lemma 3.2.

for processes that are up in this good period. In the next subsection we compute the length of a good period that is sufficient to ensure $\mathcal{P}_{su}(\,)$.

Lemma 3.2 (Timeout τ_C)**.** *Consider Parametrization 3.1 with the timeout $\tau_C = [2\Delta + (2n-1)\Phi]\beta$. Assume that a k-good period starts at time t_g and that round r_0 is the highest round started by any process in Π_0 by time t_g. Then every new round $r > r_0$ started after time t_g is space uniform ($\mathcal{P}_{su}(r)$) with cardinality k.*

Proof. We show that in round r, for every process $p \in \Pi_0$, we have: (i) p receives a message from all processes in Π_0, but (ii) not from any process not in Π_0.

We start with (ii). Assume that p received a round r message from a process q that is not in Π_0. By the definition of a good period, p could not have received this message after t_g. If p had received this message before t_g, then p would have advanced to round r immediately, which contradicts our assumption that no process in Π_0 has entered round r by t_g.

To prove (i), assume some process p_1 is the first to finish sending its round r messages at time $t_s > t_g$ (see Fig. 3.2). These messages are ready for reception at each process in Π_0 (p_2 in Fig. 3.2), the latest at $t_s + \Delta$, since messages are Δ-timely. These messages are received in the next receive step, which occurs the latest after $n-1$ send steps (in the case the process was just starting executing send steps). Since a step takes up to Φ time, p_1's message is received by all processes in Π_0 the latest at $t_s + \Delta + n\Phi$. Each process that receives this message jumps to round r, if not already there, and thus, by time $t_s + \Delta + (2n-1)\Phi$ has performed $n-1$ send steps and has sent its round r message to all. This message is ready for reception by the latest at time $t_e = t_s + 2\Delta + (2n-1)\Phi$.

The timeout $\tau_C = [2\Delta + (2n-1)\Phi]\beta$, together with Lemma 3.1, ensure that no timeout of length τ_C started at time t_s expires before t_e. So when the timeout expires, all messages for round r are either received or ready to be received. Before calling the transition function for round r, a receive step is performed; thus in round r every process in Π_0 receives a message from every process in Π_0. □

3.5.3 Length of a good period

Theorem 3.1. *In any good period of length*

$$(x+1)\left[\frac{\beta}{\alpha}\Big(2\Delta + (2n-1)\Phi\Big) + n\Phi\right] + \Delta + n\Phi$$

the generic algorithm with Parametrization 3.1 ensures x consecutive rounds that fulfill $\mathcal{P}_{su}(\,)$.

Proof. Assume a good period starts at time t_g and at this time process p_1 has the highest round number r among the processes in Π_0. We distinguish two cases: (i) t_g is during these $n-1$ send steps (not shown in Fig. 3.3) of round $r = 3\phi$. (ii) t_g after these send steps (see Fig. 3.3). It can be shown that case (ii) is worse than case (i) in terms of length of the good period, thus we consider case (ii). Round $r+1$ is the first round that all processes in Π_0 start after t_g. According to Lemma 3.2, round $r+1$, $r+2$, etc. are space uniform if the good period is long enough. We compute the maximum time it takes for any process p_2 to complete round $r+x$. As shown by Fig. 3.3, p_2 starts round $r+1$ at latest at time $t_g + \tau_U + 2n\Phi + \Delta$ (end of *"initialization"* in Fig. 3.3). This expression is obtained as follows: by the definition of p_1, no message of a round larger than r is received before p_1's timer expires, and $\tau_U = \frac{\beta}{\alpha}\tau_L$ is the time elapsed for a timeout $\tau_C = \tau_L\beta$; when the timeout expires, p_1 executes a receive step (ϕ), moves to round $r+1$, executes $n-1$ send steps ($(n-1)\Phi$); in the worst case the message to p_2 is sent in the last of these send steps; Δ later the message is ready for reception on p_2; at this time p_2 may be executing n send steps ($(n-1)\Phi$) before the reception step (Φ) in which p_1's message is finally received; at this point p_2 moves to round $r+1$.

We now show that case (i) leads to a shorter good period. Here, by time $t_g + (n-2)\Phi$, at least one message of round r was sent by p_1. By time $t_g + (n-2)\Phi + \Delta + n\Phi$, this message is received by some process, and at the latest $(n-1)\Phi$ time later, this process has sent its round r messages to all. Thus, after time $t_g + 2\Delta + (4n-4)\Phi$, every process has performed its send steps for round r. Consequently, doing now the same analysis as in case (ii), it cannot be the case anymore that a process is performing send steps when the message for round $r+1$ is ready for reception. This leads to the fact that also in this case p_2 starts round $r+1$ not after time $t_g + \tau_U + \Delta + 2n\Phi$.

Process p_2 needs at most $n\Phi + \tau_U$ to complete round $r+1$ (see *"regular round"* in Fig. 3.3): $n-1$ send steps ($(n-1)\Phi$), timeout τ_U (in the worst case no message of a larger round is received), one receive step (Φ).

Summing up the duration of "initialization" and of x "regular rounds" leads to $(x+1)[\tau_U + n\Phi] + \Delta + n\Phi$. Replacing τ_U with $\frac{\beta}{\alpha}\tau_L$, and τ_L with $2\Delta + (2n-1)\Phi$ (see Lemma 3.2) establishes the result. □

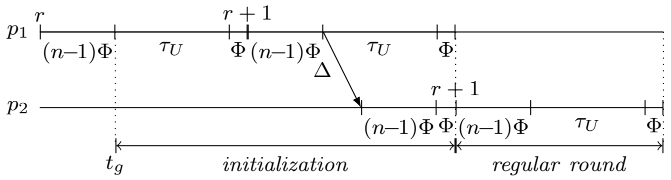
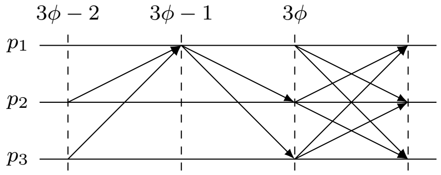


Figure 3.3: Full synchronization: length of good period - Theorem 3.1.

As already stated in Section 3.2.2, for y instances of predicate $\mathcal{P}_{otr}$ we need $2y$ space-uniform rounds, for $\mathcal{P}_{lv3}$ we need $3y+3$ space-uniform rounds, and for $\mathcal{P}_{lv4}$ we need $4y+4$ space-uniform rounds.

3.6 Phase Synchronization

Full synchronization sends extra messages with respect to the "natural message pattern" induced by the predicates $\mathcal{P}_{lv3}$ and $\mathcal{P}_{lv4}$. In this section, we give an implementation for $\mathcal{P}_{lv3}$, that uses only the "natural" messages, which are the following:



Here, processes are synchronized only at round 3ϕ of every phase ϕ. As in the case of full synchronization, round 3ϕ allows the election of the coordinator for phase $\phi+1$.

3.6.1 Outline of phase synchronization

The "natural" message pattern just depicted is generated by the function *Dest* in Parametrization 3.2.

Round 3ϕ of phase ϕ is identical to a round in the full synchronization case: all processes wait for the timeout before electing a new coordinator and moving to the first round of the next phase. According to function *NextRound*, only round $3\phi-2$ (line 2 in *NextRound*) may be terminated by the reception of messages. Rounds $3\phi-1$ and 3ϕ terminate by expiration of the timeout as before.

Round $3\phi-1$ requires some clarification. In this round processes only need to receive a message from the coordinator, thus it seems natural for a process to advance to round 3ϕ as soon as it receives such message. But this solution is not correct as shown by the following scenario. Consider

Parametrization 3.2 LV-3 using phase synchronization

$$\mathit{NextRound}(p, r, \tau, \mathit{coord}, \mathit{Rcv}) := \bigvee \begin{cases} \exists \langle -, -, r' \rangle \in \mathit{Rcv} : \; r' > r \\ r \bmod 3 = 1 \;\wedge\; (\tau \geq \tau_{C1} \;\vee\; |ho(r)| > n/2) \\ r \bmod 3 = 2 \;\wedge\; \tau \geq \tau_{C2} \\ r \bmod 3 = 0 \;\wedge\; \tau \geq \tau_{C3} \end{cases}$$

$$\mathit{SkipRound}(p, r, \tau, \mathit{coord}, \mathit{Rcv}) := \exists \langle -, -, r' \rangle \in \mathit{Rcv} : \; r' > r$$

$$\mathit{Dest}(p, r, \tau, \mathit{coord}, \mathit{Rcv}) := \begin{cases} \mathit{coord} & : \; r \bmod 3 = 1 \\ \Pi & : \; r \bmod 3 = 0 \;\vee\; (r \bmod 3 = 2 \;\wedge\; p = \mathit{coord}) \\ \emptyset & : \; \text{else} \end{cases}$$

$$\mathit{ElectCoord}(p, r, \tau, \mathit{coord}, \mathit{Rcv}) := \begin{cases} \min(\Pi) & : \; r = 1 \\ \min(ho(r-1)) & : \; r \bmod 3 = 1 \;\wedge\; ho(r-1) \neq \emptyset \\ \mathit{coord} & : \; \text{else} \end{cases}$$

processes p_1, p_2, and p_c, with p_c being the coordinator. Process p_1 receives p_c's message for round $3\phi - 1$, advances to round 3ϕ, and sends its round 3ϕ message to all. This message is delivered quickly to p_2, which receives it before the round $3\phi - 1$ message from p_c. If p_2 advances immediately to round 3ϕ, it will miss the round $3\phi - 1$ message from p_c that arrives later. Algorithm 3.2 avoids this problem by delaying the start of round 3ϕ until all processes had time to receive the round $3\phi - 1$ message from the coordinator. Another solution, based on piggybacking, is described in Section 3.6.4.

3.6.2 Timeouts τ_{C1}, τ_{C2}, τ_{C3}

Lemma 3.3 (Timeout τ_{C1}). *Consider Parametrization 3.2 with*

$$\tau_{C1} = [2\Delta + (2n+1)\Phi]\,\beta + \tau_{C3}\left(\frac{\beta}{\alpha} - 1\right).$$

Assume every process starts round $3(\phi - 1)$ in a $\left(\frac{n+1}{2}\right)$-good period and phase ϕ has a unique coordinator. Then the coordinator hears from a majority of processes in round $3\phi - 2$.

Proof. Let p_3 be the first process that starts the timeout for round $r = 3(\phi - 1)$ at time t_{s3} (see Fig. 3.4). By time $t_{s3} + \Delta + n\Phi$ all other processes, *e.g.*, p_2, are in round $3(\phi - 1)$. After sending their round $3(\phi - 1)$ messages, which takes at most $(n-1)\Phi$ time, their timeout $\tau_{C3} = \tau_{L3}\beta$ will expire by time $t_{s3} + \Delta + (2n-1)\Phi + \tau_{L3}\frac{\beta}{\alpha}$. A receive and a send step later, each process has sent its round $3\phi - 2$ message to the coordinator (p_1 in Fig. 3.4), which is by time $t_{s3} + \Delta + (2n+1)\Phi + \tau_{L3}\frac{\beta}{\alpha}$. This message is ready for reception at the coordinator Δ time later. Thus if the coordinator executes the receive step and the transition function for round $3\phi - 2$ not before time $t_{s2} = t_{s3} + 2\Delta + (2n+1)\Phi + \tau_{L3}\frac{\beta}{\alpha}$, the set of messages passed to the transition function includes the messages of round $3\phi - 2$ from a majority.

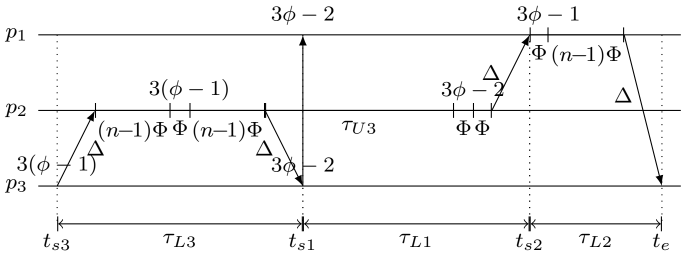


Figure 3.4: Phase synchronization: timeouts - Lemmas 3.3–3.5.

Because p_3 is the first process to start the timeout for round $3(\phi-1)$, no timeout τ_{C3} for this round expires at any process (including the coordinator) before $t_{s3} + \tau_{L3}$. Therefore, no process (including the coordinator) starts round $3\phi - 2$ before $t_{s1} = t_{s3} + \tau_{L3}$, and executes the transition function for round $3\phi - 2$ before $t_{s1} + \tau_{L1}$. The timeout τ_{C1} as given by the lemma ensures that $t_{s1} + \tau_{L1}$ is not before ts_2. Thus the coordinator p_1 receives the round $3\phi - 2$ messages from all processes in Π_0. □

Lemma 3.4 (Timeout τ_{C2}). *Consider Parametrization 3.2 with the timeout* $\tau_{C2} = [3\Delta + (3n+1)\Phi]\,\beta + \tau_{C3}\left(\frac{\beta}{\alpha} - 1\right) - \tau_{C1}$. *Assume a phase ϕ with a unique coordinator, where round $3(\phi - 1)$ starts in a $\left(\frac{n+1}{2}\right)$-good period. Then in round $3\phi - 1$, every process in Π_0 hears from the coordinator.*

Proof. Let p_3 be the first process that starts the timeout for round $r = 3(\phi - 1)$ at time t_{s3}. By a similar reasoning as for Lemma 3.3, each process has sent its round $3\phi - 2$ message to the coordinator by time $t_{s3} + \Delta + (2n + 1)\Phi + \tau_{L3}\frac{\beta}{\alpha}$. Then, at most $\Delta + \Phi$ later, the coordinator has received this message from every process in Π_0, thus achieved the majority condition in line 2 of *NextRound*, and therefore sends its round $3\phi - 1$ message to all. The latter takes at most $(n-1)\Phi$ time, and this message will be ready for reception Δ time later. At this time, $t_e = t_{s3} + 3\Delta + (3n+1)\Phi + \tau_{L3}\frac{\beta}{\alpha}$, the timeout τ_{C2} may safely expire at any process. Because p_3 is the first process that starts round $3(\phi - 1)$, no process starts round $3\phi - 1$ before $t_{s2} = t_{s3} + \tau_{L3} + \tau_{L1}$. Thus choosing τ_{C2} as in the lemma ensures that τ_{C2} does not expire before time t_e at any process. □

Lemma 3.5 (Timeout τ_{C3}). *Consider Parametrization 3.2 with the timeout* $\tau_{C3} = [2\Delta + (2n-1)\Phi]\beta$. *Assume that a k-good period starts at time t_g and that round r_0 is the highest round started by any process in Π_0 by time t_g. Then every new round $3\phi > r_0$ started after time t_g is space uniform with cardinality k.*

Proof. Similar to Lemma 3.2. □

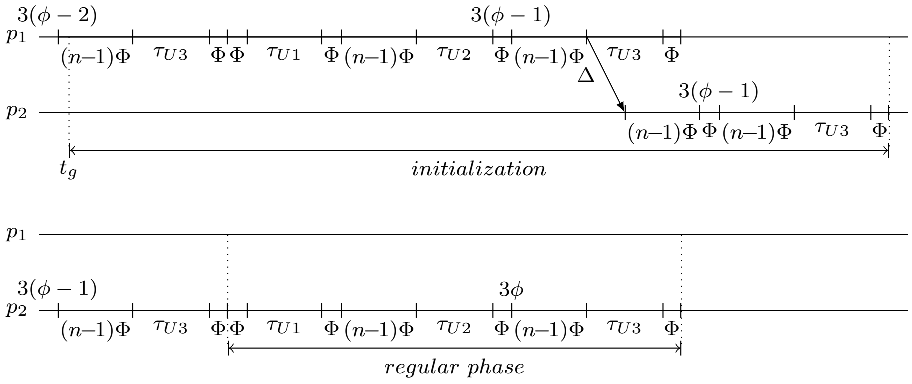


Figure 3.5: Phase synchronization: length of good period - Theorem 3.2.

Corollary 3.1. *Parametrization 3.2 with the timeout* $\tau_{C1} = 2\Phi\beta + \frac{\beta^2}{\alpha}[2\Delta + (2n-1)\Phi]$, $\tau_{C2} = [\Delta + n\Phi]\beta$, *and* $\tau_{C3} = [2\Delta + (2n-1)\Phi]\beta$ *ensures a* $\mathcal{P}_{lv3}(\phi)$, *if round* $3(\phi - 1)$ *starts in a* $\left(\frac{n+1}{2}\right)$*-good period.*

Proof. By Lemma 3.5, round $3(\phi - 1)$ is space-uniform. Thus by the definition of *ElectCoord*, every process in Π_0 has the same coordinator. Applying the Lemmas 3.3 to 3.5 and replacing the equations for τ_{C1}, τ_{C2}, and τ_{C3} yields the result. □

3.6.3 Length of a good period

Theorem 3.2. *In any good period of length*

$$y\left[\Big(2\Delta + (2n-1)\Phi\Big)\frac{\beta^2}{\alpha^2} + \Big(3\Delta + (3n+1)\Phi\Big)\frac{\beta}{\alpha} + (2n+2)\Phi\right] + \\ + \Big(2\Delta + (2n-1)\Phi\Big)\frac{\beta^2}{\alpha^2} + \Big(5\Delta + 5n\Phi\Big)\frac{\beta}{\alpha} + \Delta + 5n\Phi \quad (3.2)$$

the generic algorithm with Parametrization 3.2 and timeouts according to Corollary 3.1 ensures y consecutive phases that fulfill $\mathcal{P}_{lv3}(\phi)$.

Proof. It can be shown that the "initialization" period (see Fig. 3.5) is the longest in case t_g starts just after the first send step of some round $3(\phi-2)$, and round $3(\phi - 2)$ is not space uniform (only round $3(\phi - 1)$ is space uniform). The end of round $3(\phi - 1)$ corresponds to the end of the "initialization" period. By time $t_g+(2n+1)\Phi+\tau_{U3}+\tau_{U1}+\tau_{U2}$ round $3(\phi-1)$ is started at some process p_1. This round $3(\phi - 1)$ ends for all processes in Π_0 at the latest $(3n-1)\Phi+\tau_{U3}+\Delta$ time later (end of "initialization" period) using the same argument as in Theorem 3.1. By Corollary 3.1, we have $\mathcal{P}_{lv3}(\phi)$. Every "regular" phase then takes at most time $\tau_{U1} + \tau_{U2} + \tau_{U3} + (2n + 2)\Phi$. The result follows by replacing the timeouts with the expressions from Corollary 3.1. □

3.6.4 Piggybacking

We have explained in Section 3.6.1 why we choose round $3\phi-1$ to terminate by the expiration of a timeout. The other solution requires some changes to our generic implementation. Thus we present only the overall idea and the results.

In this approach, a process p piggybacks all the messages it received for a round r on its message for round $r+1$. If some process q receives the round $r+1$ message from p before entering round $r+1$, q can include these round r messages to its received set before ending round r. In some cases, this shortens the length of a good period.

In general, this mechanism can be used if all processes wait for the same quorum in some round, *e.g.*, in the second round of LV-3, where all processes wait for a single message from the same process, *i.e.*, the coordinator. By this optimization the length of a good period for LV-3 using phase synchronization can be reduced approximately by one Δ. The expression can be calculated by applying a similar analysis as before:

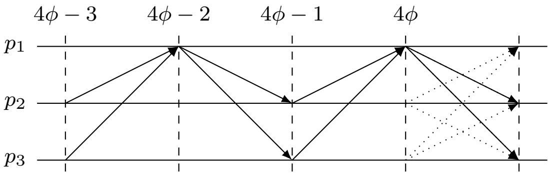
$$y\left[\Big(2\Delta+(2n-1)\Phi\Big)\frac{\beta}{\alpha}+2\Delta+(2n+2)\Phi\right]+ \\ +\Big(2\Delta+(2n-1)\Phi\Big)\frac{\beta^2}{\alpha^2}+\Big(5\Delta+5n\Phi\Big)\frac{\beta}{\alpha}+\Delta+5n\Phi \quad (3.3)$$

The other benefit of piggybacking is to speed up best case scenarios, where $\Pi_0=\Pi$ in a good period (*i.e.*, all processes are up; see Definition 3.4). In this case, the implementation of the predicate does not rely on any timeout, and the length of a good period depends only on the actual transmission delay of messages, and no more on Δ. However, applying piggybacking in every round induces an important overhead and can considerably increase the effective message transmission delay.

3.7 Synchronization by a Coordinator

If we use full synchronization or phase synchronization to implement $\mathcal{P}_{lv4}$, LV-4 will never perform better than LV-3 in terms of message complexity or length of the good period, because LV-4 requires one more round per phase. In this section we give another implementation that achieves a message complexity of $O(n)$ instead of $O(n^2)$ per regular phase during a good period, at the cost of a slightly larger length of the good period.

The predicate $\mathcal{P}_{lv4}$, contrary to $\mathcal{P}_{lv3}$, does not require any round where all processes hear from each other. Without such a round, we need to send additional messages in some round in order to synchronize processes and choose a coordinator, like we do for $\mathcal{P}_{lv3}$. As for $\mathcal{P}_{lv3}$ we do this only once per phase, in the last round of a phase. This leads to the following message pattern:



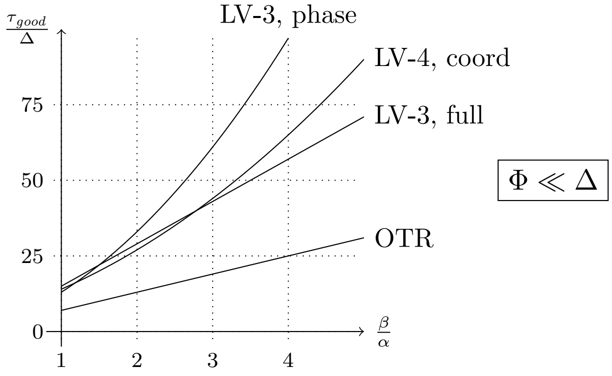
The messages represented by a full line are required by $\mathcal{P}_{lv4}$, while the messages represented by a dotted line in round 4ϕ are only for synchronization and election of a coordinator.

The complete description of this method is given in the technical report [BHSS09] and is not formally described in this thesis.

3.8 Comparison

3.8.1 Impact of clock drift

First, we analyze the impact of the clock on the duration of a good period. The following figure shows, for increasing $\frac{\beta}{\alpha}$, the dependency of the Δ term on the length of a good period τ_{good}:



With perfect clocks ($\alpha = \beta = 1$), the different implementations of the *LastVoting* algorithm lead to almost the same result, which is no more the case with large clock drift. Large clock skew occurs for instance when clocks are built from step counting. We note also that algorithms that synchronize more often (like LV-3 with full synchronization) depend less on the accuracy of clocks.

3.8.2 Analysis of the results for drift-free clocks

The observation in the previous section leads us to focus on drift-free clocks (*e.g.*, $\alpha = \beta = 1$). Table 3.1 (page 48) gives an overview of the results established in the chapter. We can analyze these results in two extreme cases: (1) short good periods (where the duration of a good period is approximately the duration of "the initialization period plus one regular phase"), and (2) long good period (where the duration of the good period is at least two orders of magnitude greater than the duration of "the initialization period plus one regular phase"). In case (2) the initialization period can be ignored; in case (1) the initialization period must be taken into account.

3.8.2.A Short good periods

With short good periods, OTR is obviously the best algorithm. However, it requires a $\left(\frac{2n+1}{3}\right)$-good period. Among the algorithms that only require a $\left(\frac{n+1}{2}\right)$-good period, LV-3 with full synchronization performs clearly the worst. However, the different implementations are close to each other.

If we assume $\Phi \ll \Delta$, then we get 7Δ for OTR and between 12Δ and 15Δ for the implementations of the *LastVoting* algorithms. If we compute the ratio between these values and the number of rounds to decide in each implementation, we get 3.5Δ per round for OTR, 4.3Δ per round for LV-3 with phase synchronization, 4Δ per round for LV-3 with piggybacking, and 3.5Δ per round for LV-4 with synchronization by a coordinator. This shows that the number of rounds per phase is not necessarily a good performance predictor.

3.8.2.B Long good periods

With long good periods, OTR is also the best algorithm, even though the relative difference with the other algorithms is smaller. Among the algorithms that only require a $\left(\frac{n+1}{2}\right)$-good period, LV-3 with full synchronization performs again the worst. However, the relative difference between the different implementation of *LastVoting* is larger than in the case of short good periods.

If we assume again $\Phi \ll \Delta$, we obtain the following ratios between the duration of a regular phase and the number of rounds to decide: 2Δ per round for OTR, 1.7Δ per round for LV-3 with phase synchronization, 1.3Δ per round for LV-3 with piggybacking, and 1.5Δ per round for LV-4 with synchronization by a coordinator. In this case, the number of rounds per phase is even a worse performance predictor than in the case of short good periods.

Finally, note that one algorithm requires only a linear number of messages per regular phase, namely LV-4 with synchronization by a coordinator.

3.8.3 Lesson

Time complexity and message complexity are the metrics generally used to measure the efficiency of (consensus) algorithms. Time complexity is usually expressed in terms of number of communication steps, or number of rounds per phase (which is equivalent, since one round requires usually one communication step).

The results obtained show that the approach developed in this chapter provides a fine analytical measure for the time complexity of consensus algorithms. Moreover, the results show that the number of communication steps is in general not a good metric for time complexity.

3.9 Conclusion

The chapter has derived the time complexity (and message complexity) for the implementation of several consensus algorithms in a system that alternates between good and bad periods. The time complexity is the duration of the good period for one instance of consensus. For each algorithm we have computed two expressions: one for the case of a short good period (that includes an initialization period plus the duration for a regular phase), and another for long good period (for which the initialization period can be ignored). Our results show that OTR has always the best time complexity, but requires a two-thirds majority of processes to be up in a good period. Among the algorithms that require only a simple majority of processes that are up, their time complexity is very close in the case of a short good period. If we consider a long good period, the difference becomes relevant. For example in the case $\Phi \ll \Delta$, which is typically the case in a WAN, LV-4 with synchronization by a coordinator has a time complexity 50% higher than LV-3 with piggybacking. However, the message complexity of the latter is higher ($n^2 + 2n$ vs. $4n$). Finally, the chapter has shown the shortcoming of the time complexity measured in number of rounds, by computing for each implementation the ratio of the two time complexity measures.

	# rounds	length of good period		# messages	
		initialization	per phase	initialization	per phase
OTR	2	$3\Delta + (4n-1)\Phi$	$4\Delta + (6n-2)\Phi$	n^2	$2n^2$
LV-3 (Full Synch)	3	$9\Delta + (13n-4)\Phi$	$6\Delta + (9n-3)\Phi$	$4n^2$	$3n^2$
LV-3 (Phase Synch)	3	$8\Delta + (12n-1)\Phi$	$5\Delta + (7n+2)\Phi$	$2n^2 + 2n$	$n^2 + 2n$
LV-3 (Piggybacking)	3	$8\Delta + (12n-1)\Phi$	$4\Delta + (4n+1)\Phi$	$2n^2 + 2n$	$n^2 + 2n$
LV-4 (Coord Synch)	4	$8\Delta + (10n-5)\Phi$	$6\Delta + (4n+2)\Phi$	$2n^2 + 3n$	$4n$

Table 3.1: Summary of results of *full*, *phase*, and *coord* synchronization for $\alpha = \beta = 1$.

Chapter 4

Swift Algorithms for Repeated Consensus

The round implementations presented in the previous chapter for consensus algorithms in the context of benign faults have a main drawback: they do not allow making progress at the speed of the system. The chapter introduces the notion of a *swift algorithm*. Informally, an algorithm that solves a repeated problem is swift, if in a partial synchronous run of this algorithm, eventually no timeout expires, *i.e.*, the algorithm execution proceeds with the actual speed of the network. This definition differs from other efficiency criteria for partial synchronous systems.

Furthermore, we show that the notion of swiftness explains the reason why failure detector based algorithms are typically more efficient than round-based algorithms, since the former are naturally swift while the later are naturally non-swift. We show that this is not an inherent difference between the models, and provide a round implementation that is swift, therefore performing similarly to failure detector algorithms while maintaining the advantages of the round model.

Publication: F. Borran and M. Hutle and N. Santos and A. Schiper. Swift Algorithms for Repeated Consensus. To appear in *the 29th IEEE International Symposium on Reliable Distributed Systems (SRDS 2010).*
The chapter is a joint work with Nuno Santos.

4.1 Introduction

Timeouts are often required to solve certain problems in distributed computing. Due to the FLP impossibility result [FLP85] and similar results for other problems, there is a need for some minimal synchrony assumptions for solving those problems, and timeouts are the dominant mechanism to make use of synchrony assumptions in an algorithm. In some cases, like in the failure detector model, timeouts are hidden (*e.g.*, in the failure detector implementation). Thus when speaking about an algorithm, we consider the overall system, in the FD case the combination of the asynchronous algorithm and the failure detector implementation that run together in a message-passing system.

Timeouts are often chosen conservatively, so that the algorithm is correct for a large number of real-life scenarios. However, timeouts should be used only to cope with faults, and not slow down the execution time in good cases. As an example, when implementing communication-closed synchronous rounds in a synchronous message passing system, after a process sends its messages for a certain round it usually waits for a timeout, before it ends the round and sends its messages for the next round. However, in many runs of the algorithm, a process might have received all messages from other alive processes already long before that. It would be favorable to start the next round immediately after all messages from alive processes are received. This is, for example, the case with an algorithm that uses a $\Diamond\mathcal{P}$ failure detector. Here, a process might wait until the FD output is accurate, which inside the FD involves waiting for a timeout, but only once: later rounds profit from the fact that the failure detector "remembers" information about faults. We formally capture such a behavior by the definition of *swift*, which we define in the context of repeated problems like repeated consensus [DGDF$^+$08]. The main intuition behind our definition is that swift algorithms are more efficient than non-swift algorithms, and are therefore preferable. A swift algorithm for a repeated problem is thus one in which eventually all problem instances are "efficient".

In more detail, for the definition of swift we look at partial synchronous runs, *i.e.*, runs where a bound Δ on the transmission delay eventually holds forever.[1] For the good period of such a run, that is the partial run R in which the bound Δ holds, we can define the actual transmission delay $\delta(R)$ as the maximum of all transmission delays in R. Such an actual transmission delay can be much smaller than the maximum transmission delay Δ. If in this case, the execution time for each instance of the repeated problem eventually depends only on $\delta(R)$ (in contrast to Δ), the algorithm is *swift*. This definition thus relates application performance (namely, execution time) to underlying system model (namely, transmission delay) metrics.

While intuitively swift algorithms progress at the speed of messages in good periods, and non swift algorithms progress only at the speed of expiration of timeouts, we refrained from calling these two classes of algorithms *message-driven* and *timeout driven*. This is because the term *message-driven* is used in [HW05, BW09] with a different meaning, namely to refer to the

[1] Note that such a run exists also, *e.g.*, in an asynchronous system, and all runs of a synchronous systems are of course also partial synchronous. The definition is thus not limited to partial synchronous systems.

way events are generated at a process. If processes are allowed to measure time (*e.g.*, with clocks or step counting), then it is possible to construct message-driven algorithms (according to this definition) that are not swift. On the other hand, if processes use an adaptive timeout, then the algorithm can be swift despite timeout expiration (see Section 4.6). Thus these terms are not suitable to precisely characterize this new class of algorithms.

Other notions of efficiency for distributed algorithms have been considered. The term *fast* has been used to refer to (consensus) algorithms that solve consensus with less communication steps in favorable cases [Sch97, HR99, MR01, Lam06]. A favorable case corresponds usually to an execution without faults that is synchronous from the beginning. On the contrary, the definition of swift is related to the execution *time* of an algorithm, in the context of *repeated* problems. Further, the definition of swift considers also runs with faults. The notion of fast is orthogonal to the notion of swift: it is possible to design both fast algorithms that are swift and fast algorithms that are not swift.

The same argument holds for *early terminating* algorithms [DRS90, Lyn96], and algorithms with *zero degradation* [DG02, DS06]. Zero degradation refers to the ability of the consensus algorithms to terminate in two communication steps in every stable run (when all failures are initial crashes, and failure detection is reliable). For early terminating consensus algorithms, the number of communication steps depends on the actual number of failures and not on the maximum number of failures.

Contribution: This chapter makes the following two contributions. The first contribution is the definition of swift algorithms that we just discussed. The second contribution is a new implementation of communication-closed rounds in a partial synchronous system with crash faults. This new implementation leads to swift round-based consensus algorithms, while previous round implementations, including those described in [DLS88, Gaf98], are not swift. This result is highly relevant in the context of comparing advantages and drawbacks of the failure detector approach [CT96] vs. the round-based approach [DLS88, CBS09], in the context of solving agreement problems. Indeed, failure detector based algorithms, despite the usage of timeouts in the implementation of the failure detector algorithm, are naturally swift. On the other hand, round implementations in a partial synchronous model have some advantages over FD based round implementations [HS07]. Our new solution thus combines the advantages of both approaches.

Roadmap: The rest of the chapter is organized as follows. In the next section, we specify our model and give a formal definition of *swift*. In Section 4.3 we start by introducing the round-based consensus algorithm as a motivating example. Then we revisit a simple implementation of rounds that is not swift, and further illustrate why a naive failure detector based solution for this problem is swift. In Section 4.5 we present our main contribution, an implementation of partial synchronous rounds that is swift. Another swift round implementation is given in Section 4.6 using an adaptive timeout mechanism. We conclude by discussing the different approaches in Section 4.7.

4.2 Definitions

4.2.1 Model

We consider a system of n processes connected by a message-passing network. Processes execute an algorithm by making steps, where a step can be either a send step $\langle p, \textsc{Send}, m\rangle$, in which a process sends a message to another process, a receive step $\langle p, \textsc{Receive}, S\rangle$, in which a (possibly empty) set S of messages is received, an input step $\langle p, \textsc{In}, I\rangle$ in which a finite (possibly empty) set of values is read from p's in-queue, or an output step $\langle p, \textsc{Out}, O\rangle$, in which a set of values is output to p's out-queue.[2] In each step a process also performs a state transition.

For simplicity we assume an abstract global discrete time from $\mathcal{T} = \mathbb{N}$. Without loss of generality, at each time $t \in \mathcal{T}$ at least one process makes a step. A single process can make at most one step at any time. Processes have no access to this time, however, in periods where some additional synchrony assumptions hold, they may measure time by counting their own steps.

Messages are unique. The message-passing system fulfills the *integrity* property defined in Section 2.1.3.

Let $\mathcal{S}_p$ denote the set of possible steps of a process p including a null step $\bot$. A run R is a function $\mathbb{N} \to \mathcal{S}_1 \times \mathcal{S}_2 \times \cdots \times \mathcal{S}_n$. Processes may fail by crashing. A process that makes an infinite number of steps in a run R is called *correct* in R, else it is called *faulty.* Let $I \subset \mathbb{N}$ be a contiguous interval. Then a *partial run* R is a function $I \to \mathcal{S}_1 \times \mathcal{S}_2 \times \cdots \times \mathcal{S}_n$. We denote with $|R| = |I|$ the length of the run measured in time. Further we define the failure pattern of a run R as $F(t) := \{p : \forall t' \geq t, R(t')[p] = \bot\}$, that is the set of all processes that take no steps after t. We extend this definition in a natural way, so that $F(R)$ denotes the faulty processes in a partial run R. Further, $\mathcal{C}(F)$ denotes the set of processes that is never in F, and $f := n - |\mathcal{C}(F)|$. Note that the distinction between correct and faulty processes and the failure pattern are needed only for the implementation that uses a failure detector (see Section 4.4).

For a run R, we denote with *In* (resp. *Out*) the in-queue (resp. out-queue) of the processes in R. The queues are functions $\Pi \times \mathcal{T} \to \Sigma^*$, where

[2] In repeated consensus, the in-queue contains the consensus proposals, and the out-queue contains the consensus decisions.

Σ is an arbitrary alphabet.

Definition 4.1 (Partial synchrony)**.** *A run R is* (Δ, Φ, W)-partial synchronous *if there is a sub-run R' with $|R'| = W$, such that the process speed bound Φ and the transmission delay bound Δ hold in R', and no process crashes in R'.*

We call the time interval that is specified by R' a *good period of R*, and denote it also as (Δ, Φ)-good period. The classic partial synchronous system [DLS88] is (Δ, Φ, ∞)-partial synchronous, since it requires a good period that lasts forever. We say a *system* is (Δ, Φ, W)*-partial synchronous* if every run R of the system fulfills Definition 4.1.

Definition 4.2 (Actual parameters)**.** *Let R be a partial run. Then $\delta(R)$ denotes the maximum transmission delay of R, i.e., the minimum value δ', such that the delay bound δ' holds in R. Further, $\phi(R)$ denotes the minimum value ϕ', such that the process speed bound ϕ' holds in R.*

If R is a good period of a (Δ, Φ, W)-partial synchronous system, then $\delta(R) \leq \Delta$ and $\phi(R) \leq \Phi$. When R is clear from the context, we simply write δ or ϕ. It should be emphasized, however, that Δ (resp. Φ) can be a *known* parameter and can be used in the code of the algorithms, while δ (resp. ϕ) is a performance metric of a single *run*, and unknown.

4.2.2 Problems

A *problem* is specified by an alphabet Σ and a predicate $P(\mathit{In}, \mathit{Out}, F)$, where In and Out are an in- resp. an out-queue over alphabet Σ, and F is a failure pattern. An *algorithm A solves a problem given by predicate P* if, for any run R where F is the failure pattern and In and Out are the in- resp. out-queue of R, predicate $P(\mathit{In}, \mathit{Out}, F)$ holds. Moreover, we define: $x \in Q$ iff x is in queue Q at some point in time, and: $x \prec y$ iff x is queued in Q before y.

Example 4.1 (Atomic broadcast)**.** For *atomic broadcast*, the following needs to hold:

(i) $\forall p \in \mathcal{C}(F), \forall m \in \mathit{In}_p : \; m \in \mathit{Out}_p$

(ii) $\forall p, q \in \mathcal{C}(F), \forall m \in \mathit{Out}_p : \; m \in \mathit{Out}_q$

(iii) $\forall p \in \mathcal{C}(F), \forall m \in \mathit{Out}_p, \; \exists q \in \Pi : \; m \in \mathit{In}_q$

(iv) $\forall p, q \in \Pi^2 : \; \{m, m'\} \subseteq \mathit{Out}_p \wedge \{m, m'\} \subseteq \mathit{Out}_q : \; m \prec m' \in \mathit{Out}_p \Leftrightarrow m \prec m' \in \mathit{Out}_q$

In this chapter, we focus on repeated problems, like repeated consensus. For such problems, every element in Σ is of form $\langle i, v \rangle$ where i is an instance number (from $\mathbb{N}$) and v the input/output value from a set V. Further, every predicate $P(\mathit{In}, \mathit{Out}, F)$ is a predicate of the form $\bigwedge_{i \in \mathbb{N}} P'(\mathit{In}_i, \mathit{Out}_i, F)$, where In_i (resp. Out_i) is In (resp. Out) restricted to elements of instance i.

Example 4.2 (Repeated consensus)**.** In the *repeated consensus* problem, for each instance i, the following needs to hold:

(i) $\forall p \in \Pi,\ \forall \langle i, v\rangle \in \mathit{Out}_p,\ \exists q \in \Pi : \langle i, v\rangle \in \mathit{In}_q$
(ii) $\forall p, q \in \Pi^2,\ \forall \langle i, v\rangle \in \mathit{Out}_p,\ \forall \langle i, v'\rangle \in \mathit{Out}_q : v = v'$
(iii) $\forall p \in \mathit{correct}(F),\ \exists v : \langle i, v\rangle \in \mathit{Out}_p$

4.2.3 Swift algorithms

Definition 4.3 (Execution time)**.** *For a run R of a repeated problem consider instance i. Let $t_{in} = \max\{t : \langle i, v\rangle$ is taken from In_i at some process p at time $t\}$. Respectively, $t_{out} = \max\{t : \langle i, v\rangle$ is output to Out_i at some process p at time $t\}$. Then the time $\tau_i(R) = t_{out} - t_{in}$ is the execution time in R of instance i of the problem.*

Let $A(\Delta, \Phi)$ be a collection of algorithms, with one algorithm for every Δ and Φ.[3]

Definition 4.4 (Swift algorithm)**.** *A collection of algorithms A that solves a repeated problem, is* swift*, if for all Φ there exist $k, c \in \mathbb{N}$ such that for every run R of $A(\Delta, \Phi)$ that is (Δ, Φ, ∞)-partial synchronous with good period R' and includes an infinite number of instances, there is an instance number i' such that for all instance $i \geq i'$, $\tau_i(R) \leq k\delta(R') + c$.*

Informally, a swift algorithm for a repeated problem is one in which eventually all instances of the problem have an execution time proportional to the actual speed of the system. Note that this definition does not refer to timeouts. Timeout expiration is a low level issue. Our definition only depends on the relation between system properties (*i.e.*, transmission delays) and algorithm properties (*i.e.*, execution time), and therefore avoids any reference to timeout expiration.

Example 4.3. Consider the following simple implementation of reliable broadcast (Example 4.1 without property (iv)) in an asynchronous system with reliable links. Every process p keeps a local instance number i_p. If p wants to broadcast a message m, sends $\langle i_p \cdot n + p, m\rangle$ to all. A process that receives this message the first time delivers this message and forwards it to all other processes. In a partial synchronous run R, every instance i that is started by reading the input m in the good period R' takes at most $(n + 2)\phi + \delta$ time (n send steps, one transmission delay, one receive step, one output step). Since $\tau_i \leq (n + 2)\phi + \delta \leq (n + 2)\Phi + \delta$ this algorithm is swift ($k = 1$ and $c = (n + 2)\Phi$, note that the constants may depend on Φ and n, see Definition 4.4).

[3] Here, Δ and Φ are just algorithm parameters. For models with known bounds on process speed and transmission delays, Δ and Φ intuitively represent this knowledge. For models with unknown bounds, or asynchronous algorithms, we assume $A(\Delta, \Phi)$ to be a constant function, giving a single algorithm.

We are now also in the position to illustrate the initial example from the introduction in more detail:

Example 4.4. Consider an implementation of *FloodSet* consensus algorithm [Lyn96] in a synchronous system with $\Phi = 1$, where each round r takes exactly $(\Delta + (n+2)\Phi)$ steps (one input step, n send steps, one transmission delay Δ, and one receive step). New inputs are read only when the previous consensus instance has finished. Any instance of consensus requires $f + 1$ rounds, so every instance decides after $(f + 1)(\Delta + (n + 2)\Phi) + \Phi$. Since $(f+1)(\Delta+(n+2)\Phi)+\Phi = k\Delta+c \not\leq k\delta+c$ in general, this implementation is not swift.

Although this natural implementation of *FloodSet* algorithm is not swift, our solution in Section 4.5 can be used to implement a swift version of *FloodSet*.

4.3 A Non-Swift Round-based Algorithm

We now explain swiftness and non-swiftness on simple consensus algorithms for partially synchronous systems in more details. The algorithms we consider belong all to the same class of consensus algorithms, *i.e.*, algorithms that require $f < n/3$.

4.3.1 Consensus algorithm

We consider a round-based algorithm, namely the *OneThirdRule* (OTR) consensus algorithm from Section 3.2, see Algorithm 4.1, expressed slightly differently. In this representation, the state of each process consists of its estimate x_p and its decision $decision_p$, initially set to $\perp$. In each round r, a process sends its estimate x_p to all processes (line 6) and then, after an implicit receive step where only messages of round r may be received, performs the state transition function (lines 8 to 11). As already discussed, Algorithm 4.1 is always safe. For liveness, we need two *space uniform* rounds (defined in Section 3.2) in which the set Π_0 of alive processes (at least $2n/3$) receives all messages from processes in Π_0, and only from these processes. This can be ensured by the round implementation layer during the good period of a partially synchronous system.

4.3.2 Round implementation

The implementation of the round structure is given by Algorithm 4.2. It is an extension of the implementation given in [HS07] and Section 3.5, with support for repeated instances of consensus.

Algorithm 4.1 *OneThirdRule* (OTR) (code of process p)

```
State:
  $x_p \in V$
  $decision_p \in V$; initially $\perp$

Round $r$ :
  $S_p^r$:
    send $\langle x_p \rangle$ to all processes
  $T_p^r$:
    if number of values received $> 2n/3$ then
      $x_p :=$ the smallest most often received value
      if more than $2n/3$ values received are equal to $v$ then
        $decision_p := v$
```

Algorithm 4.2 A simple round implementation (code of process p)

```
 1: $r_p \leftarrow 1$
 2: $next\_r_p \leftarrow 1$
 3: $Rcv_p \leftarrow \emptyset$                                  /* set of received messages */
 4: $\forall i \in \mathbb{N} :\ state_p[i] \leftarrow \perp$                        /* state of instance $i$ */

 5: while true do
 6:   $I \leftarrow input()$
 7:   for all $\langle i, v \rangle \in I$ do
 8:     $state_p[i] \leftarrow init(v)$        /* initialization of state with initial value $v$ */
 9:   for all $i :\ state_p[i] \neq \perp$ do
10:     $msgs[i] \leftarrow S_p^{r_p}(state_p[i])$
11:   for all $q \in \Pi$ do
12:     $M_q \leftarrow \{\langle i, msgs[i][q] \rangle :\ state_p[i] \neq \perp\}$
13:     send($M_q, r_p, p$) to $q$
                                                                   input & send

14:   $i_p \leftarrow 0$
15:   while $next\_r_p = r_p$ do
16:     $i_p \leftarrow i_p + 1$
17:     if $i_p \geq TO$ then
18:       $next\_r_p \leftarrow r_p + 1$
19:     receive($M$)
20:     $Rcv_p \leftarrow Rcv_p \cup M$
21:     $next\_r_p \leftarrow \max(\{r\ :\ \langle -, r, - \rangle \in Rcv_p\} \cup \{next\_r_p\})$
                                                                   receive

22:   $O \leftarrow \emptyset$
23:   for all $i :\ state_p[i] \neq \perp$ do
24:     for all $r \in [r_p, next\_r_p - 1]$ do
25:       $\forall q \in \Pi :\ M_r[q] \leftarrow m$ if $\exists M\ \langle M, r, q \rangle \in Rcv_p \wedge \langle i, m \rangle \in M$, else $\perp$
26:       $state_p[i] \leftarrow T_p^r(state_p[i], M_r)$
27:     if $\exists v \neq \perp$ s.t. $decision(state_p[i]) = v$ for the first time then
28:       $O \leftarrow O \cup \langle i, v \rangle$                /* $v$ is the decision of instance $i$ */
29:   $output(O)$
                                                                   comp. & output
30:   $r_p \leftarrow next\_r_p$
```

Each iteration of the outermost loop is composed of three parts: *input* & *send* part, *receive* part and *comp.* & *output* part. In the *input* & *send* part, the process queries the input queue for new proposals (line 6), initializes

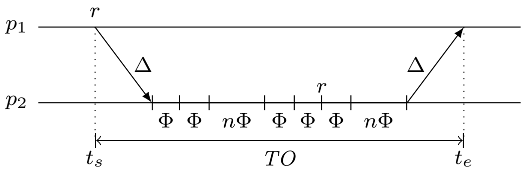


Figure 4.1: Simple round implementation: timeout $TO \geq 2\Delta + (2n + 5)\Phi$.

new slots on the *state* vector for each new proposal (line 8), calls the send function of all active consensus instances (line 10), and sends the resulting messages (line 13). The process then starts the *receive* part, where it waits for messages until either the timeout TO expires (line 17) or it receives a message from a higher round (line 21). Finally, in the *comp. & output* part, the process calls the state transition function of each active instance (line 26), and outputs any new decisions (line 29). Note that some rounds may be partially skipped (no message sent, no message received, only transition function executed): this happens whenever a message from higher round is received.

The implementation described above is not swift. Before the formal proof, we give the intuition why in each round at least one timeout has to expire. A process starts a new round either if its timeout expires (lines 17-18), or if it receives a message from a higher round (line 21). Since for the first process entering a new round, obviously no higher round message exists, it must do so necessarily by a timeout. Therefore, even in the good periods, the system can only advance as fast as the timeouts of the fastest process expire.

4.3.3 Correctness

We start by the correctness of our solution. As shown in [CBS09], Algorithm 4.1 is always safe. In order to have liveness we need a good period of a certain length:

Theorem 4.1. *Consider a run of Algorithm 4.2 with* $TO \geq 2\Delta + (2n+5)\Phi$ *and* $n > 3f$*. Let* R *be a* (Δ, Φ, W)*-partial synchronous run with good period* $[t_g, t_g + W]$ *with* $W = 3TO + \Delta + (4n+8)\Phi$*. Then every consensus instance that starts at* t *decides the latest at* $max(t, t_g) + W$*.*

The following lemmas show that in every good period of length W, there are two space-uniform rounds. Together with the results of [CBS09], this proves the theorem.

Lemma 4.1. *Consider Algorithm 4.2 with* $TO \geq 2\Delta + (2n + 5)\Phi$ *and* $n > 3f$*. Let* t_r *be the time the first process starts a new round* r*, and assume that* $[t_r, t_r + TO + (n + 1)\Phi]$ *is a* (Δ, Φ)*-good period. Then round* r *is space-uniform.*

Proof. (See Figure 4.1 for illustration.) Let p be the first process to finish the input and send steps for round r, at time t_s ($t_s \leq t_r + (n+1)\Phi$). We show that (i) all round r messages from all alive processes are ready for reception[4] by time $t_s + TO$, and (ii) no process expires its round r timeout before $t_s + TO$. This implies that round r is space-uniform.

(i) By time $t_s + \Delta$ the round r message from p is ready for reception at all processes. Every process q will make a receive step at most $(n+2)\Phi$ time later (if at time $t_s + \Delta$ q was on a output step of a round $r' < r - 1$, then it must make one input step and n send steps before the next receive step). After receiving the round r message, every process performs an output step for its current round, advances to round r, performs one input and n send steps. Therefore, by time $t_s + \Delta + (2n+5)\Phi$, all processes have finished sending their round r messages, and Δ time later, by time, $t_s + 2\Delta + (2n+5)\Phi = t_s + TO$, all round r messages are ready for reception at all alive processes. Note that this time is still in the good period, since $t_s + TO = t_r + TO + (n+1)\Phi$.

(ii) Since all processes start the timeout for round r after p, the timeout of no process will expires before $t_s + TO$. Additionally, no process advances to a higher round by receiving a higher round message because for a new round to start, the timeout of round r of some process has to expire. □

Lemma 4.2. *Consider Algorithm 4.2. Let* $[t_g, t_g + W]$ *be a* (Δ, Φ)*-good period, with* $W = TO + (n+2)\Phi$*. Then by time* $t_g + W$ *at least one process has started a new round* r_0*.*

Proof. Let p be the process with the highest round number r among all processes. Then the lemma is fulfilled, if p is at least in round $r+1$ by the given time. However, in a good period, p can be in round r at most for $TO + (n+2)\Phi$ time, the timeout and the time for an input, an output, and n send steps. □

4.3.4 Non-swiftness

To show that the algorithm is not swift, from here on we distinguish between the *a priori* known parameters Δ and Φ, which have to hold in all runs, and the effective values of $\delta(R)$ and $\phi(R)$, which are the maximum transmission delay and maximum step time of the good period R. This distinction allows us to show what part of the duration of a round or an instance is from a timeout (terms in Δ and Φ) and what part is from the time required to send a message or to perform a step (expressed in δ and ϕ).

Lemma 4.3 (Maximum execution time)**.** *Consider Algorithm 4.2 with* $TO \geq 2\Delta + (2n+5)\Phi$*,* $n > 3f$*, and a* (Δ, Φ, ∞)*-partial synchronous run with a*

[4] We call a message ready for reception if it must be received with the next receive step of the receiver process.

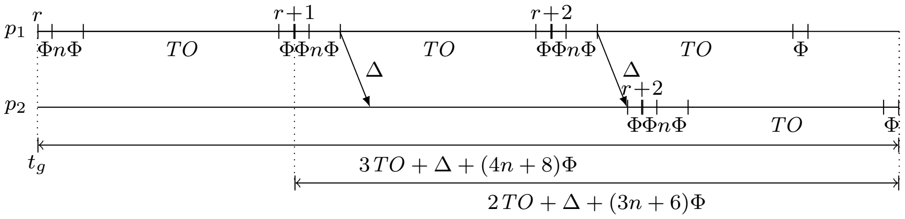


Figure 4.2: Simple round implementation: length of good period - Theorem 4.1 and Lemma 4.3.

good period R that starts at time t_g. Let r_0 be the first new round that is started after t_g. Then for all instances i started in a round $r \geq r_0$, we have an execution time $\tau_i \leq 2TO + \delta + (3n+6)\phi$.

Proof. (See Figure 4.2 for illustration.) Let i be an instance started in a round $r \geq r_0$ by a process p. Recall that Algorithm 4.2 needs at most two space-uniform rounds to decide. Since by Lemma 4.1 rounds r and $r+1$ are space-uniform, all processes decide instance i by round $r+1$ (*i.e.*, they output (i, x) at line 29, where x is the decision).

It remains to calculate the maximum time for rounds r and $r+1$. Let p be the first process to start round r at time t_r. Process p will finish round r the latest at $t_r + TO + (n+2)\phi$, and start the send steps for round $r+1$ Φ time later. By time $t_r + TO + \delta + (2n+3)\phi$, p's round $r+1$ messages are ready for reception at all processes. At this point, all processes have finished executing the send steps for round r (process p's round r messages forced them to advance) and are either executing receive steps for round r or have entered round $r+1$. Therefore, all processes will enter round $r+1$ at most ϕ time after receiving p's round $r+1$ message. Round $r+1$ will take at most $TO + (n+2)\phi$ time, so by time $t_r + 2TO + \delta + (3n+6)\phi$ all processes have finished round $r+1$. □

Since $\delta \leq \Delta$ and $\phi \leq \Phi$, Theorem 4.1 follows also from Lemma 4.3. For the minimal value of TO, the maximum execution time is $4\Delta + (4n+10)\Phi + \delta + (3n+6)\phi$.

We compute now also the minimum execution time, in order to show that the solution is not swift:

Lemma 4.4 (Minimum execution time)**.** *Consider Algorithm 4.2 with $TO \geq 2\Delta + (2n+5)\Phi$, $n > 3f$, and a (Δ, Φ, ∞)-partial synchronous run with good period R that starts at time t_g. Let r_0 be the first new round that is started after t_g. Then for all instances i started in a round $r \geq r_0$, we have an execution time $\tau_i > \Delta$.*

Proof. Assume by contradiction that for an instance i that started in a round $r \geq r_0$, we have $\tau_i \leq \Delta$. This means that there is a process p that stays in round r at most Δ time.

Let t_r and t_e be the time when p starts and finishes round r, respectively. According to the code of algorithm, process p may finish round r in two cases: either (i) by the expiration of its timeout, or (ii) by receiving a higher round message. In case (i) we have $t_e - t_r = TO + (n+2) > \Delta$. In case (ii) let q be the first process that has finished round r and sent round $r+1$ messages to all. Process q could do this only by expiration of its timeout for round r. Therefore, q has started round r the latest by time $t_q = t_e - TO - n - 4$. Process q sent a round r message to p by time $t_q + (n+1)\phi$, and p received it $\delta + (n+2)\phi$ later. Because in the worst case p is doing an output step for another round. So by time $t_q + \delta + (2n+4)\phi$, p, after an output step, must have entered round r, therefore $t_r = t_q + \delta + (2n+4)\phi$. Expanding t_r, we obtain $t_r = t_e - TO - n - 4 + \delta + (2n+4)\phi < t_e - \Delta$, that is, $t_e - t_r > \Delta$, which contradicts the assumption that p remained in round r for at most Δ. Therefore, for all $r \geq r_0$, all processes remain in round r for more than Δ time. A contradiction. □

Theorem 4.2. *The collection of algorithms $A(\Delta, \Phi)$ given by Algorithm 4.2 is not swift.*

Proof. In case that $TO < 2\Delta + (2n+5)\Phi$, the algorithm is not live. Therefore we only consider $TO \geq 2\Delta + (2n+5)\Phi$.

Assume by contradiction that the collection of algorithms $A(\Delta, \Phi)$ given by Algorithm 4.2 is swift. Then, for a fixed Φ, there exist $k, c \in \mathbb{N}$, such that in every (Δ, Φ, ∞)-partial synchronous run R with a good period R', there is an instance i_R such that, for all instances $i > i_R$, $\tau_i(R) < k\delta(R') + c$. For a contradiction, consider $A(k\delta(R') + c, \Phi)$. By Lemma 4.4, for almost all instances i started after GST, $\tau_i > \Delta = k\delta(R') + c$. A contradiction. □

4.4 A Failure Detector-based Algorithm that is Swift

We consider now the OTR algorithm expressed with the failure detector $\Diamond\mathcal{P}$, see Algorithm 4.3. Repeated execution of Algorithm 4.3 is expressed by Algorithm 4.4. The box in Algorithm 4.4 corresponds to line 7 of Algorithm 4.3. The algorithm is expressed in a model similar to the RRFD model of Gafni [Gaf98]. For simplicity, we have not shown in Algorithm 4.4 the (trivial) implementation of $\Diamond\mathcal{P}$ in a partially synchronous system. We assume that, both Algorithm 4.4 and implementation of $\Diamond\mathcal{P}$, run concurrently in the following way: in every even step, Algorithm 4.4 is executed, in every odd step, the implementation of $\Diamond\mathcal{P}$ is executed.

Intuitively it is easy to see that Algorithm 4.4 is swift. Indeed, some time after GST, the failure detector list contains exactly the faulty processes. At this point, by line 7, all correct processes wait only for messages from correct

Algorithm 4.3 OTR with the failure detector $\Diamond\mathcal{P}$ (code of process p)

```
 1: State:
 2:    $r_p \leftarrow 1$                                        /* round number */
 3:    $x_p \in V$
 4:    $decision_p \in V$

 5: while true do
 6:    send $\langle r_p,\, x_p \rangle$ to all processes
 7:    wait until received values for round $r_p$ from all processes $q \notin \Diamond\mathcal{P}_p$
 8:    if number of values received $> 2n/3$ then
 9:       $x_p \leftarrow x$ smallest most often received value
10:       if more than $2n/3$ values received are equal to $v$ then
11:          $decision_p \leftarrow v$
12:    $r_p \leftarrow r_p + 1$
```

Algorithm 4.4 Multiple instances of Algorithm 4.3 (code of process p)

```
 1: Initialization:
 2:    $r_p \leftarrow 1$
 3:    $\forall i \in \mathbb{N} :\ x_p[i] \leftarrow \bot$
 4:    $\forall i \in \mathbb{N} :\ decision_p[i] \leftarrow \bot$

 5: while true do
 6:    $I \leftarrow input()$
 7:    for all $\langle i, v \rangle \in I$ do
 8:       $x_p[i] \leftarrow v$
 9:    send $\langle r_p, x_p, p \rangle$ to all processes
10:    while not received $\langle r_p, x_q, q \rangle$ from all processes $q \notin \Diamond\mathcal{P}_p$ do
11:       $receive(M)$
12:       $Rcv \leftarrow Rcv \cup M$
13:    $O \leftarrow \emptyset$
14:    for all $i : x_p[i] \neq \bot$ and $decision_p[i] = \bot$ do
15:       if number of values received $\langle r_p, x', - \rangle > 2n/3$ then
16:          $x_p[i] \leftarrow$ smallest most often value $x'[i]$
17:          if more than $2n/3$ values $x'[i]$ are equal to $v$ then
18:             $decision_p[i] \leftarrow v$
19:             $O \leftarrow O \cup \{\langle i, v \rangle\}$
20:    $output(O)$
21:    $r_p \leftarrow r_p + 1$
```

processes, and since $f < n/3$, the condition of line 8 is always true. Note that the failure detector model requires reliable links (defined in Section 2.1.3), contrary to the solution in the previous section.[5] For simplicity we will assume that links are natively reliable in this section, although they can be implemented from eventual reliable links with a retransmission protocol. The correctness of Algorithm 4.4 follows from the following lemma:

Lemma 4.5. *For Algorithm 4.4, there is eventually a round GSR so that for all rounds $r \geq GSR$, every correct process receives a message from every*

[5] Consider two correct processes p and q and line 7 executed by p. If the message sent by q is lost, and p's failure detector never suspects q, then p is blocked forever at line 7.

correct process in round r and receives no message from faulty processes.

Proof. By the properties of $\Diamond\mathcal{P}$, there is a time where the FD is accurate and complete, *i.e.*, a process is suspected if and only if it is faulty. In every round that is started after this time, every correct process waits for a message from every correct process. □

Theorem 4.3 proves that Algorithm 4.4 is swift, by showing that eventually every consensus instance decides within $3\delta + (6n + 6)\phi$.

Theorem 4.3. *For a run of Algorithm 4.4 with $n > 3f$ and an infinite number of instances, there is an instance number i_0 such that for all instances $i \geq i_0$, we have an execution time $\tau_i \leq 3\delta + (6n + 6)\phi$.*

Proof. Let GSR be the round defined by Lemma 4.5. Since in every input step only a finite number of instances are read, there is an input step so that this step and all later input steps are just before a round that is after GSR. Let i_0 be the largest consensus instance started in a round before GSR (instance i is started in the round in which the last process starts instance i). Consider an instance $i > i_0$. The maximum execution time of instance i corresponds to the maximum duration of two rounds. This follows from Lemma 4.5 and the fact that Algorithm 4.3 requires two rounds after GSR. In this case, the algorithm decides in at most two rounds. It remains to calculate the maximum time for two rounds after GSR.

Let t be the first time a process, say p, starts round $r > GSR$. Since $r - 1 \geq GSR$, p receives round $r - 1$ messages from all correct processes. Every other process does so the latest at $t + 2(n - 1)\phi + \delta$ (note that we have to double the time for a step, since only every second step is of the asynchronous algorithm). To see this, note that every process sends its message for round $r - 1$ the latest by $t + 2(n - 2)\phi$, since p received already this message from every correct process at time t. All other processes are performing only receive steps at this time, thus these messages are received by time $t + 2(n - 1)\phi + \delta$ and all processes are in round r.

By $t + 2\delta + 2(2n)\phi$ all round r messages are thus ready for reception, and received by $t + 2\delta + 2(2n + 1)\phi$. Again by $t + 2\delta + 2(3n + 2)\phi$ all round $r + 1$ messages are sent, and thus round $r + 1$ ends the latest at $t + 3\delta + 2(3n + 3)\phi$. □

Failure detector based consensus algorithms require reliable links. This has the following implication. In contrast to the round implementation of Section 4.3, no round is skipped, *i.e.*, processes send messages for all rounds, and wait for the messages from all non-suspected processes. This implies that, unlike the round implementation in the previous section, it is no more possible to bound the time from GST until the first decision. To see this, note that at GST, a process p might be in a round r that is arbitrarily smaller than the highest round number r_{max} at that time. Since other

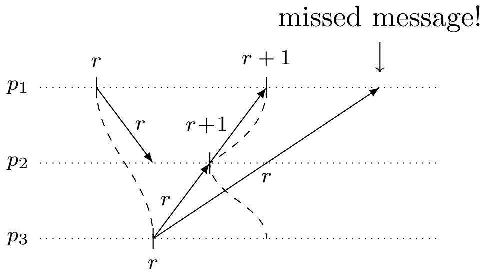


Figure 4.3: Simple solution is not correct.

correct processes might wait in any round r' ($r \leq r' \leq r_{max}$) for the round r message of process p, p cannot skip the sending of all rounds between r and r_{max}. This can also not be easily solved by packing all messages into a single one, since between the send steps p also has to perform receive steps (to receive messages from the other correct processes). This takes an unbounded amount of time, as $r_{max} - r$ can be arbitrarily large. Further, the failure detector based solution requires additional messages for the FD heartbeats, and also the overhead for ensuring (quasi-)reliable links might not be negligible.

Remark. We obtain similar results for a traditional failure detector-based consensus algorithm instead of the RRFD-based approach. Therefore, the advantages and disadvantages are not due to the round abstraction.

4.5 A New Round Implementation that is Swift

This section presents the main technical contribution of this chapter, which combines the (practical) advantages of the partial synchronous round implementation with those of implementations using failure detectors. We observe that a similar mechanism that allows the failure detector approach to be swift, can be applied to a round-based implementation, without reintroducing any additional heartbeat messages or overhead due to the reliable link assumption. Like in the failure detector approach, each process estimates a set of alive processes (that is the complimentary of the set of suspected processes) and uses this set to terminate a round earlier in good periods, namely, as soon as it receives all messages from the alive set. Contrary to the failure detector approach, the algorithm tolerates message loss, by using a timeout that expires only in bad periods. Like in the round-based implementation, processes resynchronize after message-loss by skipping rounds. Skipping rounds also allows the algorithm to decide in a bounded time after a good period starts.

4.5.1 Issue to address

Terminating a round upon receiving all messages from alive processes together with the resynchronization mechanism of Algorithm 4.2 introduces problems that do not occur in the approaches presented before. Because messages may be delivered out of (causal) order even during good periods, a process can receive a round $r+1$ message before having all round r messages. This is illustrated in Figure 4.3. In this scenario, p_3's round r message is the last one that p_2 needs to have all round r messages. Therefore, upon receiving this message, it immediately sends its round $r+1$ message to all. But process p_1 receives the round $r+1$ message from p_2 before the round r message from p_3. If p_1 goes directly to round $r+1$ when it receives the first round $r+1$ message, it will miss p_3's round r message, therefore breaking space uniformity of round r. This situation may repeat in every round, thus preventing the algorithm from deciding. In the next section, we show how we address this problem.

4.5.2 New round implementation

Algorithm 4.5 is a round implementation that is swift. This algorithm is an enhanced version of the round implementation from Section 4.3 with the following additional elements:

(i) Each process p maintains an estimation of the set of alive processes in $Alive_p$ (see line 13), and updates it every TO_A steps. TO_A is thus the timeout used to detect faulty processes.

(ii) A process goes directly to the next round if it received a message from all processes in its alive set (lines 14-15). This is the key point to make the algorithm swift.

(iii) In any case, a process goes to the next round after TO time (lines 16-17). TO is thus the timeout for a round in bad periods.

(iv) When a process receives a round message from the next round for the first time, it waits another TO_D steps until going into this round (lines 21-22). To do this, each process p maintains a variable to_p, initially set to TO (line 8) and adapted in line 22. This avoids the problem described in Section 4.5.1 and Figure 4.3.

(v) When a process receives a message from a round higher than the next round (*i.e.*, $> r_p + 1$), it immediately goes to this round (lines 19-20). This ensures a fast resynchronization of the system after a bad period.

We now show correctness of this solution (Section 4.5.3), and then swiftness (Section 4.5.4).

Algorithm 4.5 A new round implementation that is swift (code of process p)

```
1:  $r_p \leftarrow 1$
2:  $next\_r_p \leftarrow 1$
3:  $Rcv_p \leftarrow \emptyset$                              /* set of received messages */
4:  $\forall i \in \mathbb{N}:\ state_p[i] \leftarrow \bot$            /* state for instance $i$ */

5:  while $true$ do
6:     input & send                          /* lines 6-13 of Algorithm 4.2 */

       receive
7:     $i_p \leftarrow 0$
8:     $to_p \leftarrow TO$
9:     while $next\_r_p = r_p$ do
10:       $i_p \leftarrow i_p + 1$
11:       receive($M$)
12:       $Rcv_p \leftarrow Rcv_p \cup M$
13:       $Alive_p \leftarrow$ {set of processes from whom
                there is a message within last $TO_A$ steps}
14:       if $\forall q \in Alive_p : \exists \langle M_q, r_p, q \rangle \in Rcv_p$ then
15:          $next\_r_p \leftarrow r_p + 1$
16:       if $i_p \geq to_p$ then
17:          $next\_r_p \leftarrow r_p + 1$
18:       $r \leftarrow max\{r\ :\ \langle -, r, - \rangle \in Rcv_p\}$
19:       if $r > r_p + 1$ then
20:          $next\_r_p \leftarrow r$
21:       if there is a message from round $r_p + 1$ for the first time then
22:          $to_p \leftarrow \min\{i_p + TO_D, TO\}$

23:    comp. & output                        /* lines 22-29 of Algorithm 4.2 */
24:    $r_p \leftarrow next\_r_p$
```

4.5.3 Correctness

Algorithm 4.1 together with Algorithm 4.5 solves repeated consensus in a partial synchronous system. As already discussed, Algorithm 4.1 is always safe (with $n > 3f$). Before proving that the round implementation given by Algorithm 4.5 provides liveness, we show some properties of the algorithm after GST.

When the good period starts at GST, processes will synchronize to the same round using the following two mechanisms: (i) when a process receives a higher round message, it advances either immediately (line 20), or within TO_D (lines 21-22), or when the original timeout TO expires; (ii) in any case, processes remain in a round at most TO time, starting a new round when this timeout expires (lines 16-17 and line 22). Therefore, shortly after GST, there will be a process p that starts a new round r that is higher than any round started by the other alive processes. When the other processes receive the round r message from p, they will advance to round r and send their own messages. These messages are then received by all alive processes, resulting in a space uniform round.

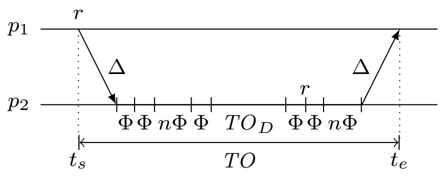


Figure 4.4: Swift round implementation: timeout $TO \geq TO_D + 2\Delta + (2n + 5)\Phi$.

As discussed in Section 4.5.1, a round $r + 1$ message may be received before all round r messages (Figure 4.3). To address this issue, if a process p in round r receives a message from round $r+1$ for the first time and it has not received all the messages from its alive set, it does not advance immediately. Instead, it waits either for an additional TO_D or until the end of the original timeout, whichever comes first. During the good period, all the remaining round r messages will be received before this updated timeout expires. To see why, notice that for a process to send a round $r+1$ message, it must have received all round r messages from the alive processes, so these messages will also be received by process p within at most $TO_D = \Delta + (n - 1)$, namely $(n-1)$ send steps and at most Δ for the transmission delay. In any case, all messages will be received before the original round timeout, so the process only has to wait for the minimum of TO_D or what is left of TO.

If a process p in round r receives a message from round $r+2$ or higher, it can conclude that the good period has not yet been started, so it advances immediately to round $r + 2$. To see why, consider that, inside the good period, process q sends a round $r + 2$ messages. Then either (i) q received all round $r + 1$ messages, including p's message, which is not possible; or (ii) the timeout for round $r+1$ expires, which is not possible as the timeout is chosen in a way that processes have enough time to receive all round messages and messages are not lost in the good period. Therefore, we are still in the bad period.

Theorem 4.4 (Correctness). *Consider a run of Algorithm 4.5 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)\Phi$, $TO \geq TO_D + 2\Delta + (2n+5)\Phi$, and $TO_A \geq TO + \Delta + (2n+1)\Phi$. Let R be a (Δ, Φ, W)-partial synchronous run with good period $[t_g, t_g + W]$ with $W = TO_A + 2TO + TO_D + 3\Delta + (6n + 15)\Phi$. Then every consensus instance that starts at t decides the latest at $max(t, t_g) + W$.*

The proof is based on the following two lemmas:

Lemma 4.6 (Timeouts TO and TO_D). *Consider Algorithm 4.5 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)\Phi$, $TO \geq TO_D + 2\Delta + (2n+5)\Phi$. Let t_r be the time at which the first process starts a new round r, and assume that $[t_r, t_r + TO + (n+1)\Phi]$ is a (Δ, Φ)-good period. Let all processes have the same alive set in this time interval. Then round r is space-uniform.*

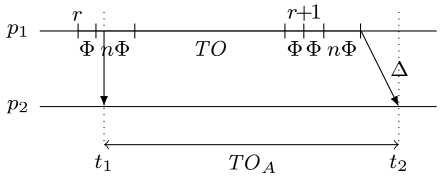


Figure 4.5: Swift round implementation: timeout $TO_A \geq TO + \Delta + (2n+1)\Phi$.

Proof. Let p_1 be the first process to finish sending its round r messages at time $t_s = t_r + (n+1)\Phi$, and starting the timeout for round r (see Figure 4.4). These messages are ready for reception at most Δ time later, at $t_s + \Delta$. These messages are received in the next receive step, which occurs the latest after $(n+2)\Phi$ steps (an output step followed by an input step, and n send steps). This is because some process (p_2 in Figure 4.4) might be just started executing an output step for some round $r' < r$. Therefore, p_1's message is received by all processes the latest at time $t_1 = t_s + \Delta + (n+3)\Phi$. Any process that receives this message in round $r-1$ for the first time, might set its timeout to $t_1 + TO_D < TO$ (see lines 21-22). And start round r the latest by time $t_1 + TO_D + \Phi$, after an output step for round $r-1$. By time $t_2 = t_1 + TO_D + \Phi + \Phi + n\Phi$, any process (including p_2) has performed an input step and n send steps for round r. This message is ready for reception the latest at time $t_e = t_2 + \Delta = t_s + TO_D + 2\Delta + (2n+5)\Phi$. The timeout $TO = TO_D + 2\Delta + (2n+5)\Phi$ ensures that no timeout started at time t_s expires before t_e (see line 16). So when the timeout expires, all messages for round r are either received or ready to be received. Before, calling the transition function for round r (in line 23), a receive step is performed (in line 11); thus every process in round r receives a message from every process, and round r is space uniform.

Note that no process in round r can receive a message from round $> r+1$. We prove this by contradiction. Let p be a process in round r that receives a message from round $r+2$. This means that there is some process q that sent round $r+2$ messages. This requires that either (i) q receives all round $r+1$ messages, including p's message, which is not possible; or (ii) the timeout for round $r+1$ expires, which is not possible inside the given interval.

If a process ends round r at time t before the end of timeout TO, because it has received all round r messages from its alive set (line 15), any other process does so the latest by time $t + (n-1)\Phi + \Delta$. From lines 21-22, a process in round r that receives a message from round $r+1$ for the first time, waits until $t + TO_D$ time before starting round $r+1$, which is enough to receive all round r messages. By the assumption, since all processes have the same actual alive set in the given interval, round r is also space uniform in this case. □

Lemma 4.7 (Timeout TO_A)**.** *Consider Algorithm 4.5 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)\Phi$, $TO \geq TO_D + 2\Delta + (2n+5)\Phi$, and $TO_A \geq TO + \Delta + (2n+1)\Phi$. Let t_r be the time the first process starts a new round r, and assume that $[t_r, t_r + 2\Phi + TO_A]$ is a (Δ, Φ)-good period. Then by time $t_r + 2\Phi + TO_A$ all processes have the same alive set.*

Proof. By time $t_1 = t_r + 2\Phi$ the first round r message can be received (see Figure 4.5). From the code of the algorithm, every process starts a new round the latest every $TO + (n+2)\Phi$ steps: one input step followed by n send steps, TO receive steps followed by an output step. From the fact that a process sends at most one message in each step, every process p_1 sends messages to any process p_2 every $TO + (n+2)\Phi + (n-1)\Phi$ steps. Since a message can take at least 0 and at most Δ time to be received, every process receives a message every $x = TO + (2n+1)\Phi + \Delta$ time. From the code of the algorithm, process p_2 excludes process p_1 from its alive set, if it does not receive a message within TO_A steps (see line 13). Comparing TO_A with x we have $TO_A = x$, which is sufficient to receive a message from any alive process. □

We now can prove the main theorem (*i.e.*, Theorem 4.4):

Proof. The proof is illustrated by Figure 4.6 (page 74). We distinguish two cases (1) $t < t_g$, (2) $t \geq t_g$. In case (1) by Lemma 4.6 a new round is started the latest at $t_g + TO + (n+2)\Phi$. From Lemma 4.7 all processes have the same alive set by time $t_0 = t_g + TO + TO_A + (n+4)\Phi$. In case (2) by Lemma 4.7 all processes have the same alive set by time $t_0 = t_g + TO_A + 2\Phi$, which is strictly smaller than $t_g + TO + TO_A + (n+4)\Phi$. From the code of the algorithm a process, *e.g.*, p_1, starts a new round r every $TO + (n+2)\Phi$ steps, *i.e.*, the latest by time $t_1 = t_0 + TO + (n+2)\Phi$. All processes do so by time $t_2 = t_1 + TO_D + \Delta + (2n+5)\Phi$. This means that all processes start round r with the same alive set. From Lemma 4.6, round r is space uniform. Furthermore, all processes receive all round r messages from their alive set, and end round r the latest by time $t_3 = t_2 + \Delta + (n+2)\Phi$ and start round $r+1$ at this time.

From the assumption, no process crashes in W, therefore, the alive set remains the same in the interval $[t_g + TO_A, t_g + W]$. In round $r+1$, all processes send their messages to all the latest by time $t_3 + (n+1)\Phi$. These messages can be received by all processes the latest by time $t_3 + (n+1)\Phi + \Delta$. From lines 14-15, all processes end round $r+1$ the latest by time $t_3 + (n+1)\Phi + \Delta + \Phi$ after an output step. This means that all processes decide the latest by this time which is equal to $t_0 + TO + (TO_D + 2\Delta + (4n+9)\Phi) + (\Delta + (n+2)\Phi)$ or $max(t, t_g) + TO_A + 2TO + TO_D + 3\Delta + (6n+15)\Phi$. □

4.5.4 Swiftness

In order to show that Algorithm 4.1 together with the round implementation provided by Algorithm 4.5 is swift, we show that the execution time of a consensus instance depends only on δ and not on Δ.

The main properties of the algorithm related to the swiftness, which hold after GST, are the following. First, the *Alive* set becomes accurate the latest by $GST + TO + (n+2)\Phi + TO_A + 2\Phi$ (line 13). Then, once the alive set is accurate after GST, it no more changes and therefore no further timeout expires. Finally, all processes finish rounds as soon as all messages from alive processes are received and advance round by lines 14-15, making the algorithm swift.

Theorem 4.5 (Swiftness)**.** *Consider Algorithm 4.5 with $n > 3f$ and the following timeouts: $TO_D \geq \Delta + (n-1)\Phi$, $TO \geq TO_D + 2\Delta + (2n+5)\Phi$, and $TO_A \geq TO + \Delta + (2n+1)\Phi$. Consider a (Δ, Φ, ∞)-partial synchronous run with a good period R that starts at time t_g. Then every consensus instance that is started after $t_g + X$, with $X = TO_A + 2TO + TO_D + 2\delta + (5n+13)\phi$, has an execution time of $\tau_i \leq 3\delta + (4n+7)\phi$.*

Proof. From Lemma 4.6 a new round starts in the good period by time $t_g + TO + (n+2)\Phi$ From Lemma 4.7 all processes have the same alive set by time $t_0 + TO_A + 2\Phi$. From the code of the algorithm a process, *e.g.*, p_1, starts a new round r every $TO + (n+2)\Phi$ steps, *i.e.*, the latest by time $t_1 = t_0 + TO_A + TO + (n+4)\Phi$. All processes do so by time $t_2 = t_1 + TO_D + \delta + (2n+5)\Phi$. This means that all processes start round r with the same alive set. From Lemma 4.6, round r is space uniform. Furthermore, all processes receive all round r messages from their alive set, and end round r the latest by time $t_3 = t_2 + \delta + (n+2)\Phi$ and start round $r+1$ at this time. Therefore, all processes end the first space-uniform round the latest by time $t_g + TO_A + TO + TO_D + 2\delta + (4n+9)\phi$. This proves the first part of the theorem with $X = TO_A + 2TO + TO_D + 2\delta + (5n+13)\phi$.

Similar to the proof of Theorem 4.3, there are instances that start after $t_g + X$. It remains to calculate the execution time for a consensus instance that is started after $t_g + X$. Since no process crashes in the (Δ, Φ, ∞)-good period, the alive set remains the same (see line 13). Let t be the first time process p_1 starts a new round r. Then every other processes does so $(n-1)\phi + \delta + (n+4)\phi$ time later. This is because some process p_2 might send the last round $r-1$ message $(n-1)\phi$ steps later to another process, *e.g.*, p_3. And p_3 will start round r the latest after $(n+4)\phi$ steps (an output followed by an input, n send, one receive, and one output step). Therefore, by time $t + \delta + (2n+3)\phi$ all processes start round r. By time $t + 2\delta + (3n+4)\phi$, all round r messages can be received, and round r ends after an output step by time $t + 2\delta + (3n+5)\phi$. Again by time $t + 2\delta + (4n+6)\phi$ all round $r+1$ messages are sent, and thus round $r+1$ ends the latest at time

$t + 3\delta + (4n + 7)\phi$. □

Since $\delta \leq \Delta$ and $\phi \leq \Phi$, the second part of Theorem 4.4 follows from Theorem 4.5.

4.6 A Swift Round Implementation using an Adaptive Timeout

The idea of the swift round implementation proposed in the previous section cannot be easily adapted to Byzantine faults. The reason is that the *Alive* set never becomes accurate in the presence of Byzantine processes. In this section we show another swift round implementation that can be adapted for Byzantine consensus algorithms (this approach is used in Chapter 6). It uses an adaptive timeout mechanism to find out the actual transmission delay. For simplicity, we do not consider step counting here. We assume that processes have access to a non-synchronized (drift-free) local clocks, by which they can measure time. In this case $\delta(R)$ denotes the maximum *end-to-end transmission delay* of run R.

The intuitive idea of the algorithm is the following: Each process starts the first phase with an initial timeout Γ_0 per round. We assume that Γ_0 is much smaller than the actual end-to-end transmission delay (*i.e.*, $\Gamma_0 \ll \delta$). If a process is not able to finish all started consensus instances with the given timeout, it increases the timeout for the next phase. In order to tolerate asynchrony periods, processes reset their timeout to Γ_0 if there is no process that needs a larger timeout.

Algorithm 4.6 is the pseudo-code of the swift round implementation using an adaptive timeout. Each process p keeps a round number r_p and a view number v_p, initially equal to 1. While the round number corresponds to the round number of the consensus algorithm, the view number increases only upon reconfiguration, *i.e.*, the timeout is a function of the view number. In addition to round number, the view number is also attached to each message (see line 17). Each process keeps track of the start time of each consensus instance (line 12). The algorithm is started with an initial round timeout Γ_0 (line 18). Messages are received until the timeout expires (line 20). When a process receives a message from a higher round or view, it updates its round or view (see lines 25 and 26). The *comp. & output* part is the same as previous (except the view number). We assume that the consensus algorithm that uses the round implementation decides in α rounds in the best case. If at the end of α rounds, there is a consensus instance in which some process p did not decide, then p starts a new view (line 39) in which it doubles the timeout. Once a process has decided for all started instances, it asks to reset the timeout, as well as the view number, by sending a Reset message (line 41). If $f + 1$ processes ask to reset the view number and

Algorithm 4.6 A swift round implementation using adaptive timeout with $n > 2f$ (code of process p)

```
 1: $r_p \leftarrow 1$                                          /* round number */
 2: $next\_r_p \leftarrow 1$
 3: $Rcv_p \leftarrow \emptyset$                                /* set of received messages */
 4: $\forall i \in \mathbb{N}: \ state_p[i] \leftarrow \bot$    /* state of instance $i$ */
 5: $\forall i \in \mathbb{N}: \ start_p[i] \leftarrow 0$      /* starting round for instance $i$ */
 6: $v_p \leftarrow 1$                                          /* view number */
 7: $next\_v_p \leftarrow 1$

 8: while $true$ do
                                                                input & send
 9:   $I \leftarrow input()$
10:   for all $\langle i, v \rangle \in I$ do
11:     $state_p[i] \leftarrow init(v)$        /* initialization of state with initial value $v$ */
12:     $start_p[i] \leftarrow r_p$
13:   for all $i: \ state_p[i] \neq \bot$ do
14:     $msgs[i] \leftarrow S_p^{r_p}(state_p[i])$
15:   for all $q \in \Pi$ do
16:     $M_q \leftarrow \{\langle i, msgs[i][q] \rangle : \ state_p[i] \neq \bot\}$
17:     send($M_q, v_p, r_p, p$) to $q$

                                                                receive
18:   $timeout_p \leftarrow current\_time + \Gamma(v_p)$
19:   while $next\_v_p = v_p$ and $next\_r_p = r_p$ do
20:     if $current\_time \geq timeout_p$ then
21:       $next\_r_p \leftarrow r_p + 1$
22:     receive($M$)
23:     $Rcv_p \leftarrow Rcv_p \cup M$
24:     $maxr \leftarrow \max(\{r \ : \ \langle -, -, r, - \rangle \in Rcv_p\})$
25:     $next\_r_p \leftarrow \max(maxr, next\_r_p)$
26:     $next\_v_p \leftarrow \max(\{v \ : \ \langle -, v, maxr, - \rangle \in Rcv_p\} \cup \{next\_v_p\})$
27:     if exists $f+1$ processes $q$ s.t. $\langle \text{RESET}, 1, -, q \rangle \in Rcv_p$ then
28:       $next\_v_p \leftarrow 1$

                                                                comp. & output
29:   $O \leftarrow \emptyset$
30:   for all $i: \ state_p[i] \neq \bot$ do
31:     for all $r \in [r_p, next\_r_p - 1]$ do
32:       $\forall q \in \Pi: \ M_r[q] \leftarrow m$ if $\exists M \ \langle M, v_p, r, q \rangle \in Rcv_p \wedge \langle i, m \rangle \in M$, else $\bot$
33:       $state_p[i] \leftarrow T_p^r(state_p[i], M_r)$
34:     if $\exists v \neq \bot$ s.t. $decision(state_p[i]) = v$ for the first time then
35:       $O \leftarrow O \cup \langle i, v \rangle$          /* $v$ is the decision of instance $i$ */
36:   $output(O)$

37:   if $v_p > next\_v_p \ \wedge \ next\_r_p \bmod \alpha = 1$ then
38:     if $\exists i: start_p[i] \leq next\_r_p - \alpha \ \wedge \ decision(state_p[i]) = \bot$ then
39:       $next\_v_p \leftarrow \max(next\_v_p, v_p + 1)$
40:     if $\forall i: start_p[i] \leq next\_r_p - \alpha \ \wedge \ decision(state_p[i]) \neq \bot$ then
41:       send($\text{RESET}, 1, next\_r_p, p$)
42:   $r_p \leftarrow next\_r_p$
43:   $v_p \leftarrow next\_v_p$
```

timeout, the view number will be reset to 1 (line 28) and timeout to Γ_0 (line 18).

Properties of Algorithm 4.6: The main properties of Algorithm 4.6 are listed below:

1. If one correct process starts round r (resp. view $v > 1$) at time τ, then all correct processes start round r (resp. view v) by time $\tau + \delta$ (lines 25-26).
2. If a correct process p starts round r (of view v) at time τ, it will start round $r + 1$ the latest by time $\tau + \Gamma(v)$ (lines 20-21).
3. A timeout $\Gamma(v) \geq 2\delta$ for round r ensures that if a correct process starts round r (of view v) at time τ, it receives all round r messages from all correct processes before the expiration of the timeout (by time $\tau + 2\delta$). From item 1, if a correct p starts round r (of view v) at time τ, then all correct process start round r (of view v) by time $\tau + \delta$. The round r (of view v) message takes another δ time. Therefore by time $\tau + 2\delta$, all round r (of view v) messages are received.
4. If all correct processes want to reset the view number, then all correct processes eventually reset the view number (because of line 27 and $n - f \geq f + 1$).

We assume $\Gamma(v) = 2^{v-1}$, *i.e.*, the timeout doubles in each new view until $2^{v-1}\Gamma_0 \geq 2\delta$. In other words, the timeout doubles until reaching view v_0 such that $2^{v_0} \geq \frac{4\delta}{\Gamma_0}$. Therefore, we have (with $\alpha = 2$ for the OTR algorithm):

$$\tau_i = \sum_{v=1}^{v_0} \alpha(2^{v-1}\Gamma_0) = 2(2^{v_0} - 1)\Gamma_0 < 8\delta.$$

This implies that the algorithm is swift, although it is not as efficient as the previous swift round implementation.

4.7 Conclusion

In this section, we sum up the differences between the three round implementations presented in this chapter, Section 4.3-4.5 (see Table 4.1). Although they all provide equivalent round abstractions, they are quite different in their performance. The last round implementation (presented in Section 4.6) uses different system model and is not comparable with other results.

The first solution, namely using a direct round implementation in the partial synchronous model, is not swift but has some advantages over the second one that is based on a failure detector:

- it does not require reliable links and does not suffer the overhead to simulate reliable links.
- it does not use any heartbeat messages but only application messages, so there is no additional message overhead.

Implementation	Simple [HS07] (Section 4.3)	FD-based [Gaf98] (Section 4.4)	Our solution (Section 4.5)
Link assumption	lossy, $\Diamond$ timely	reliable, $\Diamond$ timely	lossy, $\Diamond$ timely
Messages	only app. msgs	additional heartbeat msgs	only app. msgs
Decision time	bounded	unbounded	bounded
Execution time[6]	$4\Delta + \delta + O(\Phi)$	$3\delta + O(\phi)$	$3\delta + O(\phi)$
Swiftness	no	yes	yes

Table 4.1: Comparing different round implementations for benign faults.

- after a good period starts, the algorithm always decides in a known bounded time (Decision time in Table 4.1).

The FD based solution, however, is swift without any additional effort, while not sharing the advantages mentioned above. Our solution combines the advantages of both approaches and gives a round implementation so that the resulting repeated consensus algorithm is swift, while maintaining all the advantages above. Our last solution shows a swift round implementation that can be extended for Byzantine faults.

Note that our solution is not limited to a specific consensus algorithm. In particular, it is also possible to use the same round implementation together with the *LastVoting* algorithm [CBS09], a round-based variant of Paxos [Lam98].

[6] For simplicity, we omitted the constants in terms of Φ and ϕ.

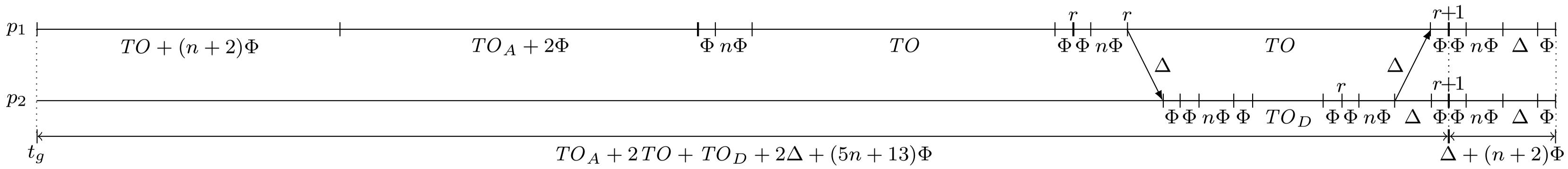

p_1
p_2
$TO + (n+2)\Phi$
$TO_A + 2\Phi$
Φ $n\Phi$
TO
r
r
Φ Φ $n\Phi$
Δ
TO
Δ
r+1
Φ Φ $n\Phi$ Δ Φ
Φ Φ $n\Phi$ Φ TO_D
r
Φ Φ $n\Phi$ Δ
r+1
Φ Φ $n\Phi$ Δ Φ
t_g
$TO_A + 2TO + TO_D + 2\Delta + (5n+13)\Phi$
$\Delta + (n+2)\Phi$

Figure 4.6: Swift round implementation: length of good period - Theorem 4.4.

Part II

Byzantine Faults

Chapter 5

Decentralized Byzantine Consensus Algorithm

The chapter considers the consensus problem in a partially synchronous system with Byzantine faults. It turns out that, in the partially synchronous system, all deterministic algorithms that solve consensus with Byzantine faults are leader-based. This is not the case of benign faults, which raises the following fundamental question: is it possible to design a deterministic Byzantine consensus algorithm for a partially synchronous system that is not leader-based? The chapter gives a positive answer to this question, and presents a decentralized (non-leader-based) algorithm that is resilient-optimal and signature-free.

Publication: F. Borran and A. Schiper. A Leader-free Byzantine Consensus Algorithm. *The 11th International Conference on Distributed Computing & Networking (ICDCN 2010)*: 67–78. Brief announcement at *the 23rd International Symposium on Distributed Computing (DISC 2009)*: 479-480.

5.1 Introduction

In this chapter we consider the Byzantine consensus problem in a partially synchronous system. The consensus algorithms proposed in [DLS88] for a partially synchronous system, for both benign and Byzantine faults, achieve safety in all executions, while guaranteeing liveness only if there exists a period of synchrony. Recently, several papers have considered the partially synchronous system model for Byzantine fault-tolerant (BFT) protocols [CL02, ACKM06, MA06, KAD$^+$07, ACKL08].

However, [ACKL08] points out a potential weakness of these protocols, namely that they can suffer from "performance failure". According to [ACKL08], a performance failure occurs when messages are sent slowly by a Byzantine leader, but without triggering protocol timeouts, and the paper points out that the Castro-Liskov leader-based PBFT algorithm [CL02] is vulnerable to such an attack. Similar arguments are mentioned in [CWA+09] and in [Lam09], where Lamport suggests the use of a virtual leader. In fact, Byzantine protocols whose progress is driven by messages from a large number of correct processes, *e.g.*, [BO83, MNCV06], are less vulnerable to performance degradation. Voting in such protocols mask performance failures, because no collection of faulty processes can prevent the correct processes from moving forward.

Interestingly, all deterministic Byzantine fault-tolerant algorithms for non-synchronous systems are leader-based, *e.g.*, [DLS88,CL02,MA06,KAD+07]. Note that these protocols use leader not only for ordering the requests during the normal executions, but also for reaching agreement during the recovery phase (see [MHS09] for details). Therefore, the Byzantine consensus algorithms used by the existing BFT protocols are leader-based.

Indeed in PBFT, t consecutive Byzantine leaders, say l_1, l_2, ..., l_t could do construct the following attack. The first leader l_1 is mute, the timeout expires, the recovery protocol is activated, and the algorithm switches to the next leader (rotating coordinator) while doubling the timeout. The same happens for leaders l_2 to l_{t-1} until l_t becomes leader. The last leader l_t sends its message as late as possible, but not too late to remain leader. If l_t remains leader forever, then the time required for any request (instance of consensus) is high.

Even the protocol in [ACKL08] is leader-based. However, the authors of [ACKL08] managed to make the leader-based protocol less vulnerable to performance failure attacks than PBFT [CL02] through a complicated mechanism that enables non-leader processes to (i) aggressively monitor the leader's performance, and (ii) compute a threshold level of acceptable performance. Note that randomized consensus algorithms such as [BO83, Rab83] are not leader-based. This raises the following fundamental question: is it possible to design a deterministic Byzantine consensus algorithm for a partially synchronous system that is not leader-based? With such an algorithm, performance failure of Byzantine processes might be harmless.

One may imagine that decentralized algorithms (algorithms that do not use the notion of a leader) for benign faults might be extended for Byzantine faults. A decentralized algorithm typically consists of a sequence of rounds, where in each round all processes send messages to all, and a correct process updates its value based on the values received. It is not difficult to design an algorithm based on this all-to-all communication pattern that does not violate the validity and agreement properties of consensus, even with Byzantine faults. However, termination requires that in some round r all correct processes receive exactly the same set of messages (from correct *and* from faulty processes). Let us denote this property for round r by $uniform(r)$. Indeed, if $uniform(r)$ holds and each correct process applies a deterministic function to the received values, the configuration becomes univalent. Can we ensure the existence of a round r in which *uniform*(r) holds?

For benign faults, it is easy to guarantee that during the synchronous period of the partially synchronous system, in every round r, all correct processes receive messages from the same set of processes. This is not the case for Byzantine faults. In round r, a Byzantine process could send a message to some correct process, and no message to some other correct process. If this happens, *uniform*(r) does not hold. Therefore one may think that with Byzantine faults the leader is needed to ensure termination, and conclude that no deterministic decentralized Byzantine consensus algorithm could exist in a partially synchronous system. We show that this intuition is wrong.

Contribution The idea of our algorithm is the following. We started from the observation that decentralized consensus algorithms exist for the synchronous system, both for benign faults (*e.g.*, the *FloodSet* algorithm [Lyn96]) and for Byzantine faults (*e.g.*, the algorithm based on interactive consistency [PSL80]). However, these algorithms violate agreement if executed during the asynchronous period of a partially synchronous system. Therefore we combine these algorithms with a second algorithm that never violates agreement in an asynchronous system. This methodology turned out to be successful, and the resulting decentralized Byzantine consensus algorithm, is presented here. The algorithm requires $3t + 1$ processes and does not rely on digital signatures.

Although BFT protocols do not assume a round-based model as we do in this work, the performance failure attack is possible in the case of a leader-based protocol implemented in the round-based model, in the case the round-based model is constructed on top of a partially synchronous model with unknown bounds. However, we believe that this is not the case for decentralized algorithms, *i.e.*, performance failure attacks are not effective in this case. The intuition is that, once the timeout of a correct process becomes large enough to receive all messages from correct processes, Byzantine processes cannot introduce an attack that forces the correct process to double its timeout. The next chapter validates this intuition analytically and shows under which conditions decentralized algorithms outperform leader-based algorithms.

Roadmap: The rest of the chapter is structured as follows. We define the problem and the system model in Section 5.2. Then we present our methodology to derive a decentralized consensus algorithm for Byzantine faults in Section 5.3. A simpler algorithm that uses digital signatures is proposed in Section 5.4. Finally, we conclude the chapter in Section 5.5.

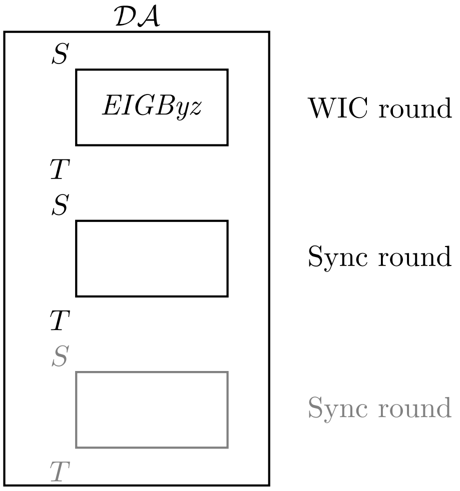


Figure 5.1: An overview of the decentralized algorithm.

5.2 System Model

We consider a set Π of n processes, among them at most t can be faulty. We solve the Byzantine consensus problem introduced in Section 2.4.2 in a partially synchronous system. We consider the Byzantine fault model with and without authentication as defined in Section 2.1.2. We use a round model abstraction on top of the system model defined in Section 2.3 which simplifies the presentation.

5.3 Byzantine Faults: From Synchrony to Partial Synchrony

In this section we explain our methodology to design a decentralized consensus algorithm that tolerates Byzantine faults (no signatures). We start with a decentralized consensus algorithm for Byzantine faults in a synchronous system model, and then extend it to a decentralized consensus algorithm in a partially synchronous system using two other algorithms.

An overview of our decentralized algorithm ($\mathcal{DA}$) is depicted in Figure 5.1. It consists of 2 or 3 rounds per phase; each round has a sending function S and a transition function T (see Section 2.2 for details). The algorithm that solves Byzantine consensus in synchronous system (called *EIGByz*), and consists of $t + 1$ rounds, is executed in the first round (WIC round). *EIGByz* constructs an eventually consistent vector to be used by the transition function of the first round of our algorithm. Other rounds of the decentralized algorithm are normal rounds that need to be eventually synchronous (Sync round), see Section 2.3 (definition of Sync and WIC

Algorithm 5.1 *EIGByz* with $n > 3t$ (code of process p)

```
Initialization:
  $W_p := \{\langle\lambda, m_p\rangle\}$          /* $m_p$ is the input value of process $p$; $val_p(\lambda) = m_p$ */

Round $r$:                                              /* $1 \le r \le t+1$ */
  $S_p^r$:
    send $\{\langle\alpha, v\rangle \in W_p \ : \ |\alpha| = r-1 \wedge p \notin \alpha \wedge v \neq \bot\}$ to all processes
  $T_p^r$:
    for all $\{q \mid \langle\alpha, v\rangle \in W_p \wedge |\alpha| = r-1 \wedge q \in \Pi \wedge q \notin \alpha\}$ do
      if $\langle\beta, v\rangle$ is received from process $q$ then
        $W_p := W_p \cup \{\langle\beta q, v\rangle\}$                      /* $val_p(\beta q) = v$ */
      else
        $W_p := W_p \cup \{\langle\beta q, \bot\rangle\}$                      /* $val_p(\beta q) = \bot$ */
    if $r = t+1$ then
      for all $\langle\alpha, v\rangle \in W_p$ from $|\alpha| = t$ to $|\alpha| = 1$ do
        $W_p := W_p \setminus \langle\alpha, v\rangle$                   /* replace $val_p(\alpha)$ ... */
        if $\exists v'$ s.t. $|\langle\alpha q, v'\rangle \in W_p| \ge n - |\alpha| - t$ then
          $W_p := W_p \cup \langle\alpha, v'\rangle$                   /* ... with $newval_p(\alpha)$ */
        else
          $W_p := W_p \cup \langle\alpha, \bot\rangle$                   /* ... with $newval_p(\alpha)$ */
      for all $q \in \Pi$ do                                  /* level 1 of the tree */
        $\vec{M}_p[q] := v$ s.t. $\langle q, v\rangle \in W_p$
```

rounds).

5.3.1 Decentralized consensus algorithm for a synchronous system

One of the first consensus algorithms that tolerates Byzantine faults in synchronous systems was proposed by Pease, Shostak and Lamport [PSL80]. It is based on an algorithm that solves the *interactive consistency* problem (see Section 2.4.3).

The algorithm presented in [PSL80] is not leader-based, does not require signatures, tolerates $t < n/3$ Byzantine faults, and consists of $t+1$ rounds of exchange of messages. We briefly recall the principle of this algorithm which is based on a *information gathering* stage followed by a *reduction function* stage (see Algorithm 5.1).

Information gathering: The information maintained by each process during the algorithm can be represented as a tree (called *Exponential Information Gathering (EIG)* tree in [Lyn96, AW04]), in which each path from the root to a leaf contains $t+2$ nodes. Thus the height of the tree is $t+1$. The nodes of each tree are labeled with sequences of processes' identities in the following manner. The root is labeled with the empty sequence λ ($|\lambda| = 0$). Let i be an internal node in the tree with label $\alpha = p_1 p_2 \dots p_r$ (see Figure 5.2); for every $q \in \Pi$ such that $q \notin \alpha$, node i has one child

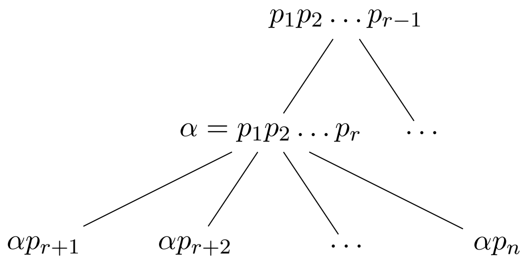


Figure 5.2: The tree construction.

labeled αq. Node i with label α will be simply called "node α". Intuitively, $val_p(p_1p_2 \dots p_r)$ (which denotes the value of node $p_1p_2 \dots p_r$ in p's tree) represents the value v that p_r told p at round r that p_{r-1} told p_r at round $r-1$ that ... that p_1 told p_2 at round 1 that p_1's initial value is v. Each correct process p maintains the tree using a set W_p of pairs $\langle$*node label*, *node value*$\rangle$. At the beginning of round r, each process p sends the $(r-1)$th level of its tree to all processes (line 5). When p receives a message from q in format $\langle p_1p_2...p_r, v\rangle$, it adds $\langle p_1p_2...p_rq, v\rangle$ to its set W_p (line 9). If p fails to receive a message it expects from process q, p simply adds $\langle p_1p_2 \dots p_rq, \perp\rangle$ to its set W_p (line 11).

Reduction function: Information gathering as described above continues for $t+1$ rounds, until the entire tree has been filled in. At this point the second stage of local computation starts. Every process p applies to each subtree a recursive data reduction function to compute a new value (lines 13 to 18). The value of the reduction function on p's subtree rooted at a node labeled α is denoted $newval_p(\alpha)$. The reduction function is defined for a node α as follows.

- If α is a leaf, its value does not change ($newval(\alpha) = val(\alpha)$);
- Otherwise, if there exists v such that $n - |\alpha| - t$ children have value v, then $newval(\alpha) = v$, else $newval(\alpha) = \perp$ (lines 16 and 18).

The reason for a quorum of size $n - |\alpha| - t$ can be explained as follows.[1] Each correct process, at the end of round $t+1$, has constructed a tree with $t+2$ levels. Any node in level $0 < k < t+1$ has $n-k$ children and a label α such that $|\alpha| = k$. If α is a label with only correct processes, then all its children except t (*i.e.*, $n-k-t$ children) have the same value.

At the end of round $t+1$, every correct process p constructs a vector $\vec{M}_p$ of size n (corresponding to level 1 of its tree), where $\vec{M}_p[q]$ is the new value of process q (line 20). *EIGByz* ensures that:

[1] Since $n > 3t$, this quorum can be replaced by $\frac{n+t}{2} - |\alpha|$ (see [PSL80]).

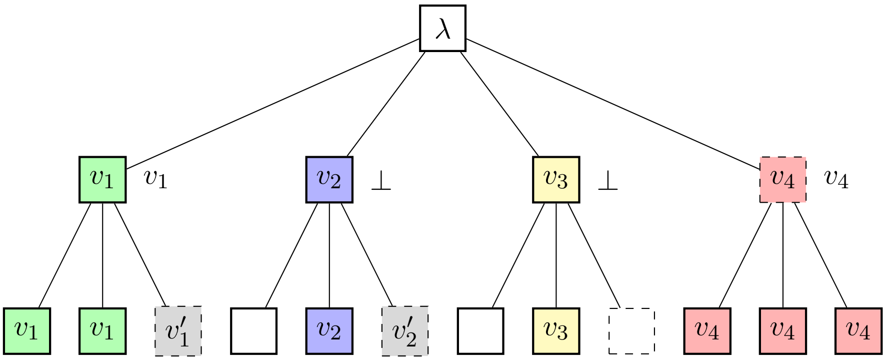


Figure 5.3: The tree constructed by a correct process in an asynchronous period with $t = 1$, $n = 4$. The value received from a correct process is shown by a solid square. The value received from a Byzantine process is shown by a dashed square. For a given node α, the value inside the square represents $val(\alpha)$ and the value beside the square represents $newval(\alpha)$.

- The correct processes compute exactly the same vector, *i.e.*, $\forall p, q \in \mathcal{C} : \vec{M}_p = \vec{M}_q$, and
- The element of the vector corresponding to a given correct process q is the input value of that process, *i.e.*, $\forall p, q \in \mathcal{C} : \vec{M}_p[q] = m_q$.

Therefore, a correct process can decide by applying a deterministic function on its vector $\vec{M}_p$.

Theorem 5.1. EIGByz *algorithm solves the interactive consistency problem for n processes with t Byzantine processes if $n > 3t$.*

Proof. The proof is given in [PSL80, Lyn96]. □

Therefore the *EIGByz* algorithm ensures the following property:

$$(\forall r,\ 1 \leq r \leq\ t{+}1 : \mathcal{P}_{Sync}(r)) \Rightarrow \forall p, q \in \mathcal{C} : (\vec{M}_p = \vec{M}_q) \wedge (\vec{M}_p[q] = m_q) \quad (5.1)$$

where $|\vec{M}_p|$ denotes the number of non-$\perp$ elements in vector $\vec{M}_p$, and m_q denotes the input value of the correct process q. The premise holds if the system is synchronous. Equation 5.1 is equivalent to the WIC round definition from Section 2.3.

5.3.2 Decentralized Consensus Algorithm for a Partially Synchronous System

If Algorithm 5.1 is executed in a partially synchronous system, it does not ensure $\forall p, q \in \mathcal{C} : (\vec{M}_p = \vec{M}_q) \wedge (\vec{M}_p[q] = m_q)$. Therefore, it cannot ensure the agreement property of Byzantine consensus. However, the following two properties hold for Algorithm 5.1 in synchronous as well as in asynchronous periods (see Figure 5.3 for illustration):

$$\forall p, q \in \mathcal{C} : \vec{M}_p[q] \in \{m_q, \bot\} \tag{5.2}$$

$$\forall q \in \Pi \setminus \mathcal{C}, \exists v \text{ s.t. } \forall p \in \mathcal{C} : \vec{M}_p[q] \in \{v, \bot\} \tag{5.3}$$

where m_q is the input value of process q. The first property (5.2) is equivalent to the property $\mathcal{P}_{Int}$ defined in Section 2.3.

To ensure agreement in a partially synchronous system, we need to combine Algorithm 5.1 with another algorithm. We show below two such algorithms: (i) a simple algorithm (Algorithm 5.2), which requires $n > 5t$, see Section 5.3.2.A, and (ii) a more complex algorithm with optimal resilience $n > 3t$ (Algorithm 5.3), see Section 5.3.2.B. In both cases, Algorithm 5.2 and Algorithm 5.3 ensure agreement, while Algorithm 5.1 ensures termination.

Before presenting the algorithms, we prove the properties of *EIGByz* algorithm. We first show two lemmas (adapted from [Lyn96] for a synchronous system) that hold in any execution of Algorithm 5.1, both in synchronous and asynchronous periods.

Lemma 5.1. *Let q be a correct process, and p some other correct process such that $val_p(\alpha q) \neq \bot$. Then, after $t+1$ rounds, for all correct processes p' we have $val_{p'}(\alpha q) = val_p(\alpha q)$ or $val_{p'}(\alpha q) = \bot$.*

Proof. If $q \notin \{p, p'\}$, then the result follows from the fact that, since q is correct, it sends the same message to p and p' at round $|\alpha|+1$. If the message sent by q to p' gets lost, then $val_{p'}(\alpha q) = \bot$. If $q \in \{p, p'\}$, then the result follows similarly from the convention by which each process relays values to itself. □

Lemma 5.2. *Let q be a correct process, and p some other correct process such that $val_p(\alpha q) \neq \bot$. Then, after $t+1$ rounds we have $newval_p(\alpha q) = val_p(\alpha q)$ or $newval_p(\alpha q) = \bot$.*

Proof. By induction on the tree labels, working from the leaves up - that is, from those of length $t+1$ down to those of length 1.

Basis: Suppose, αq is a leaf, that is, $|\alpha q| = t+1$. Then Lemma 5.1 implies that all correct processes p have the same $val_p(\alpha q)$ or $\bot$. Then also $newval_p(\alpha q) = val_p(\alpha q)$ or $newval_p(\alpha q) = \bot$ for every correct process p, by the definition of *newval* for leaves.

Inductive step: Suppose $|\alpha q| = r$, $1 \leq r \leq t$. Then Lemma 5.1 implies that for all correct processes p, have the same $val_p(\alpha q)$, call this v, or $\bot$. Therefore, every correct process q' send the same value v for αq to all processes at round $r+1$, so $val_p(\alpha qq') = v$ or $val_p(\alpha qq') = \bot$ for all correct p and q'. Then the inductive hypothesis implies that also $newval_p(\alpha qq') = v$ or $newval_p(\alpha qq') = \bot$ for all correct processes p and q'.

We now claim that $newval_p(\alpha q) = v$ or $newval_p(\alpha q) = \bot$. The number of children of αq is exactly $n-r$ which is $\geq n-t$ (i). At most t of the children have labels ending with faulty processes. Since $n > 3t$ we have $n-r-t > t$ (ii). From (i), (ii) and the definition of *newval* we have $newval_p(\alpha q) = v$ or $newval_p(\alpha q) = \bot$. □

Based on these two lemmas we prove the following lemmas. Lemma 5.3 proves (5.2), while Lemma 5.4 proves (5.3).

Lemma 5.3. *Let q be a correct process with input value m_q, and p some other correct process. Then, after $t+1$ rounds, we have $\vec{M}_p[q] = m_q$ or $\vec{M}_p[q] = \bot$.*

Proof. Assume that q is a correct process with input value m_q. We have to show that for any correct process p, $\vec{M}_p[q] \in \{m_q, \bot\}$ or $newval_p(q) \in \{m_q, \bot\}$. First note that from Lemma 5.1 when $|\alpha| = 0$, and for some process p, $val_p(q) = m_q$, then for all correct process p', $val_{p'}(q) \in \{m_q, \bot\}$. Then from Lemma 5.2 when $|\alpha| = 0$, for some correct process p, $val_p(q) = m_q$, then $newval_p(q) \in \{m_q, \bot\}$ or $\vec{M}_p[q] \in \{m_q, \bot\}$. □

Lemma 5.4. *Let q be a faulty process. There exists v such that after $t+1$ rounds, for all correct processes p, we have $\vec{M}_p[q] = v$ or $\vec{M}_p[q] = \bot$.*

Proof. Assume that q is a faulty process. And some correct process p has $\vec{M}_p[q] = v$ or $newval_p(q) = v \neq \bot$. We have to show that for all correct processes p', $newval_{p'}(q) \in \{v, \bot\}$. $newval_p(q) = v$ means that the node labeled q in the tree constructed by correct process p has at least $n-1-t$ children labeled qx with $newval_p(qx) = v$ because of the *newval* definition (i). However, since node q is a faulty process, among its children, only $t-1$ of them have a label ending with faulty process (ii). We denote $Q = \{q' \mid newval_p(qq') = v \ \wedge \ q' \text{is correct}\}$. From (i) and (ii) we have $|Q| \geq n-1-t-t+1 = n-2t$. And $\forall q' \in Q : newval_p(qq') = v$. From Lemma 5.2 we have $\forall q' \in Q : val_p(qq') = v$. From Lemma 5.1, for any correct process p' we have $\forall q' \in Q : val_{p'}(qq') \in \{v, \bot\}$. Again from Lemma 5.2 we have $\forall q' \in Q : newval_{p'}(qq') \in \{v, \bot\}$. This holds for at least $n-2t$ of node q's children in tree constructed by p'. So at most $2t-1$ of node q's children might have $newval_{p'}(qq') = v' \notin \{v, \bot\}$. Since $n > 3t$, we have $n-1-t$ (the required quorum) $> 2t-1$ which means that v' cannot be a *newval* and for all correct processes p' we have $\vec{M}_{p'}[q] \in \{v, \bot\}$. □

5.3.2.A Decentralized consensus algorithm $\mathcal{A}_1$ with $n > 5t$

We start with a simple parameterized consensus algorithm (see Algorithm 5.2). Parametrization allows us to easily adjust the algorithm to ensure agreement for different fault models. The parameterized version was first given

Algorithm 5.2 $\mathcal{A}_1$ with $n > 5t$ (code of process p)

```
1:  Initialization:
2:     $x_p := v_p \in V$                                  /* $v_p$ is the initial value of $p$ */

3:  Round $r = 2\phi - 1$:                                    /* WIC round */
4:     $S_p^r$:
5:        send $\langle x_p \rangle$ to all processes
6:     $T_p^r$:
7:        if number of non-$\bot$ elements in $\vec{\mu}_p^r > T$ then
8:           $x_p :=$ smallest most frequent non-$\bot$ element in $\vec{\mu}_p^r$

9:  Round $r = 2\phi$:
10:    $S_p^r$:
11:       send $\langle x_p \rangle$ to all processes
12:    $T_p^r$:
13:       if more than $E$ elements in $\vec{\mu}_p^r$ are equal to $v \neq \bot$ then
14:          DECIDE($v$)
```

in [BCBG$^+$07] to tolerate "corrupted communication". Here, since we consider "Byzantine process faults" we need different values for the parameters.

The algorithm consists of a sequence of phases ϕ, where each phase has two rounds $2\phi - 1$ and 2ϕ. Round 2ϕ is a normal round; to ensure termination, round $2\phi - 1$ is the round that should be eventually synchronous and consistent, *i.e.*, WIC round (defined in Section 2.3). Each process p has a single variable x_p, and in every round p sends x_p to all processes. Parameter T (line 7) refers to a "threshold" for updating x_p, and parameter E (line 13) refers to "enough" same values to decide.[2]

The algorithm was first presented in [CBS09] as *OneThirdRule* algorithm, for $T = E = 2n/3$, to tolerate $t < n/3$ benign faults. With Byzantine faults, $\mathcal{A}_1$ ensures agreement with $E \geq (n+t)/2$ and $T \geq 2n - 2E + 2t$. Strong validity requires $T \geq 2t$ and $E \geq t$. Termination, together with a WIC round provided by *EIGByz*, requires $n - t > T$ and $n - t > E$. Putting all together, for the case $E = T$, we get $T = E = 2(n+t)/3$ and $n > 5t$.

We will refer to $\mathcal{A}_1$ with $T = E = n - t$ and $n > 5t$ as *OneFifthRule* algorithm. The algorithm of Martin and Alvisi [MA06] which is called the *Fast Byzantine Paxos* can be expressed using the parameterized algorithm with $T = n - t$, $E = (n + 3t)/2$ and $n > 5t$ (see [MHS09] for more details). This version of the Martin and Alvisi algorithm is much simpler than the original algorithm which was expressed using "proposers", "acceptors", and "learners". Moreover, it satisfies the strong validity property of Byzantine consensus instead of the weak validity.

Proofs of $\mathcal{A}_1$: We first discuss agreement and strong validity of $\mathcal{A}_1$. Figure 5.4 illustrates how agreement holds. A correct process p receives 5

[2] The notation $\vec{\mu}_p^r$ is introduced in Section 2.3.

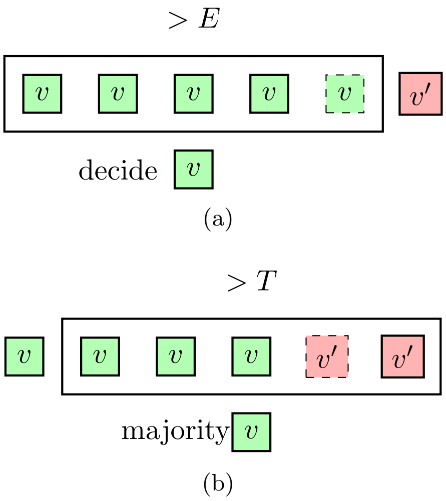


Figure 5.4: Illustration for agreement of $\mathcal{A}_1$ with $t = 1$, $n = 6$. The value received from a correct process is shown by a square. The value received from a Byzantine process is shown by a dashed square.

values v in round r (1 from Byzantine process) and decides v as shown in Figure 5.4(a). Byzantine process has sent v in round r but v' in round $r+1$. Another correct process p' can receive 2 values equal to v' in round $r+1$ as shown in Figure 5.4(b). However, p' cannot choose v' as a most frequent value after receiving 5 values.

We now give the lemmas.

Lemma 5.5. *Consider $\mathcal{A}_1$ with Byzantine faults and $E \geq \frac{n+t}{2}$. If some correct process decides v in phase ϕ, then some other correct process can only decide v in phase ϕ.*

Proof. Assume that some correct process p decides v in round $r = 2\phi$. From condition at line 13, p has received more than E values v in round r, *i.e.*, more than $\frac{n+t}{2} - t$ correct processes have sent v in round r. This means that at most $n - \frac{n+t}{2} + t = \frac{n+t}{2}$ processes could have sent a value $v' \neq v$ in round r. Since $E \geq \frac{n+t}{2}$, value v' cannot be decided in round r. □

Lemma 5.6. *With Byzantine faults and $T \geq 2n - 2E + 2t$, if some correct process decides v in round $r = 2\phi$ of $\mathcal{A}_1$, every correct process q that updates x_q in round $r' > r$, sets it to v.*

Proof. Assume that some correct process p decides v in round $r = 2\phi$. First we prove by induction on r that more than $E - t$ correct processes q have $x_q = v$ in round $r' \geq r$.

Base step ($r' = r$): Since p decides v in round r (line 14), from condition at line 13, p receives more than E values v in round r, *i.e.*, more than $E - t$ correct processes q have $x_q = v$ in round r.

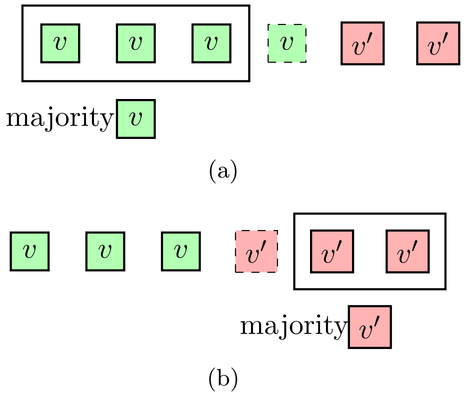


Figure 5.5: Illustration for termination of $\mathcal{A}_1$ with $t = 1$, $n = 6$ ($v' < v$).

Induction step (from $r' = 2\phi'$ to $r' + 1$): By induction hypothesis, more than $E - t$ correct processes q have $x_q = v$ in round $r' > r$. Therefore, at most $n - E + t$ processes q' might send $x_{q'} = v' \neq v$ in round $r' + 1$. A correct process q updates x_q only in line 8, and if it receives messages from $k > T$ processes. From the assumption we have $T \geq 2n - 2E + 2t$ or $k > 2(n - E + t)$. Therefore, no correct process q updates x_q to v' in round $r' + 1$. This implies that more than $E - t$ correct processes q have $x_q = v$ in round $r' + 1$.

Let q' be some correct process that updates $x_{q'}$ in some round $r' = 2\phi' - 1 > r$. Since more than $E - t$ correct processes q have $x_q = v$ in round r', and $T \geq 2n - 2E + 2t$, by same arguments as in induction step, q' sets $x_{q'}$ to v in round r'. □

Lemma 5.7. *With Byzantine faults, $T \geq 2t$ and $E \geq t$, if all correct processes p have $x_p = v$ in round $r = 2\phi - 1$ of $\mathcal{A}_1$, every correct process q that updates x_q in round $r' \geq r$, sets it to v.*

Proof. Assume that all correct processes have the same initial value v. Consider some correct process p so that the condition at line 7 holds. This means that p has received more than T non-$\perp$ messages. From $T \geq 2t$, p has received at least $2t + 1$ non-$\perp$ messages. Among these messages at most t can have a value $v' \neq v$, and at least $t + 1$ messages have v. Therefore, if p updates x_p at line 8, it sets x_p to v. □

We discuss now termination. Note that in the context of Byzantine faults, $\mathcal{A}_1$ without a WIC round does not ensure termination. Figure 5.5 illustrates the problem. A Byzantine process sends v to three first processes, and v' to two last processes in round $r = 2\phi - 1$. Assuming $v' < v$, v is the most frequent value for the first group and v' is the smallest most frequent value for the second group. This shows the need for a WIC round.

For termination, it is sufficient for $\mathcal{A}_1$ to have one round $r = 2\phi - 1$ in which the following holds (where $|\vec{\mu}_p^r|$ denotes the number of non-$\bot$ elements in vector $\vec{\mu}_p^r$):

$$\forall p, q \in \mathcal{C} : (\vec{\mu}_p^r = \vec{\mu}_q^r) \wedge (|\vec{\mu}_p^r| > T) \tag{5.4}$$

and one round $r + 1 = 2\phi$ in which we have:

$$\forall p \in \mathcal{C} : |\vec{\mu}_p^{r+1}| > E. \tag{5.5}$$

If (5.4) holds, all correct processes set x_p to some common value v_0 in round r (line 8), and if (5.5) holds all correct processes decide v_0 in round $r + 1$ (line 14).

By comparing (5.4) and (5.5) with the definitions of Section 2.3, it is easy to see that a WIC round ensures (5.4) and a synchronous round ensures (5.5), if $|\mathcal{C}| > T$ and $|\mathcal{C}| > E$ (where $|\mathcal{C}| = n - t$).

Therefore we have the following theorem.

Theorem 5.2. *With Byzantine faults, $n > 5t$ and $T = E = 2(n + t)/3$, if round $r = 2\phi - 1$, with $r \geq GSR$, is a WIC round, then $\mathcal{A}_1$ ensures strong validity, agreement and termination.*

Proof. Agreement follows directly from Lemmas 5.5, 5.6. Strong validity follows from Lemma 5.7. For termination, if (5.4) holds, all correct processes set x_p to the some common value v_0 in round r (line 8), and if (5.5) holds all correct processes decide v_0 in round $r+1$. By comparing (5.4) and (5.5) with definition of a WIC round and *GSR* it is easy to see that $\mathcal{A}_1$ ensures (5.4) and (5.5) if $|\mathcal{C}| \geq n - t$, $n - t > T$, $n - t > E$. □

Note that, the *OneThirdRule* algorithm ($\mathcal{A}_1$ with $T = E = 2n/3$) cannot be used with Byzantine faults because of the agreement problem. Using a WIC round, two correct processes cannot receive two different values from a Byzantine process in a single round, however, this can happen in distinct rounds (see Figure 5.4) which prevents us from using *OneThirdRule* algorithm.

5.3.2.B Decentralized consensus algorithm $\mathcal{A}_2$ with $n > 3t$

As $\mathcal{A}_1$ requires $n > 5t$, its resilience is not optimal. Here we show a new algorithm, which uses mechanisms from several consensus algorithms, *e.g.*, Ben-Or [BO83], and PBFT [CL02] with strong validity, and requires only $n > 3t$ (see Algorithm 5.3). Note that, as for $\mathcal{A}_1$, $\mathcal{A}_2$ ensures strong validity and agreement, but not termination. As for $\mathcal{A}_1$ termination is ensured using a first WIC round provided by *EIGByz*.

$\mathcal{A}_2$ consists of a sequence of phases ϕ, where each phase has three rounds ($3\phi - 2$, $3\phi - 1$, 3ϕ). Each process p has an estimate x_p, a vote value $vote_p$

Algorithm 5.3 $\mathcal{A}_2$ with $n > 3t$ (code of process p)

```
Initialization:
  $x_p := v_p \in V$                                /* $v_p$ is the initial value of $p$ */
  $pre\text{-}vote_p := \emptyset$
  $vote_p \in V \cup \{?\}$, initially ?
  $ts_p := 0$

Round $r = 3\phi - 2$:                                          /* WIC round */
  $S_p^r$:
    send $\langle x_p, vote_p \rangle$ to all processes
  $T_p^r$:
    if at least $n - t$ elements in $\vec{\mu}_p^r$ are equal to $\langle -, ? \rangle$ then
      $x_p :=$ smallest most frequent element $\langle x, - \rangle$ in $\vec{\mu}_p^r$
      $pre\text{-}vote_p := pre\text{-}vote_p \cup \{\langle x_p, \phi \rangle\}$
    if at least $n - t$ elements in $\vec{\mu}_p^r$ are equal to $\langle v, - \rangle$ then
      $pre\text{-}vote_p := pre\text{-}vote_p \cup \{\langle v, \phi \rangle\}$

Round $r = 3\phi - 1$:
  $S_p^r$:
    send $\langle v \mid \langle v, \phi \rangle \in pre\text{-}vote_p \rangle$ to all processes
  $T_p^r$:
    if at least $n - t$ elements in $\vec{\mu}_p^r$ are equal to $\langle v \rangle$ then
      $vote_p := v$; $ts_p := \phi$; $x_p := v$

Round $r = 3\phi$:
  $S_p^r$:
    send $\langle vote_p, ts_p, pre\text{-}vote_p \rangle$ to all processes
  $T_p^r$:
    if at least $2t + 1$ elements in $\vec{\mu}_p^r$ are equal to $\langle v \neq ?, \phi, - \rangle$ then
      DECIDE($v$)
    if exists $\langle v \neq ?, ts, - \rangle$ in $\vec{\mu}_p^r$ s.t. $vote_p \neq v$ and $ts > ts_p$ then
      if exists $t + 1$ elements $\langle -, -, pre\text{-}vote \rangle$ in $\vec{\mu}_p^r$ s.t. $\langle v, ts' \rangle \in pre\text{-}vote$ and
      $ts' \geq ts$ then
        $vote_p := ?$; $ts_p := 0$; $x_p := v$
    if $vote_p \neq ?$ then $x_p := vote_p$
```

(initially ?), a timestamp ts_p attached to $vote_p$ (initially 0), and a set *pre-vote*$_p$ of valid pairs $\langle vote, ts \rangle$ (initially $\emptyset$). The structure of the algorithm is as follows:

- If a correct process p receives the same estimate v in round $3\phi - 2$ from $n - t$ processes, then it accepts v as a valid vote and puts $\langle v, \phi \rangle$ in *pre-vote*$_p$ set. The pre-vote set is used later to detect an invalid vote.
- If a correct process p receives the same pre-vote $\langle v, \phi \rangle$ in round $3\phi - 1$ from $n - t$ processes, then it votes v (*i.e.*, $vote_p = v$) and updates its timestamp to ϕ (*i.e.*, $ts_p = \phi$).
- If a correct process p receives the same vote v with the same timestamp ϕ in round 3ϕ from $2t + 1$ processes, it decides v.

The algorithm guarantees that (i) two correct processes do not vote for different values in the same phase ϕ; and (ii) once $t+1$ correct processes have the same vote v and the same timestamp ϕ, no other value can be voted in the following phases. Before giving the formal proofs we discuss agreement and termination.

Agreement of $\mathcal{A}_2$: A configuration is v-valent if (i) $\exists \phi$ such that at least $t+1$ correct processes p have $(vote_p, ts_p) = (v, ts)$ with $ts \geq \phi$, and (ii) the other correct processes q have $(vote_q, ts_q) = (v' \neq v, ts')$ with $ts' < \phi$.

Let ϕ_0 be the smallest round in which some correct process decides v (line 26). By line 25 at least $t+1$ correct processes p have $vote_p = v$, $ts_p = \phi_0$, and $x_p = v$ from line 20; the other correct processes q with $vote_q \neq v$ have $ts_q < \phi_0$ from line 19. Therefore the v-valent definition holds. We denote the former set by $\Pi_{=\phi_0}$, and the latter by $\Pi_{<\phi_0}$. Processes in $\Pi_{=\phi_0}$ keep $x_p = vote_p = v$ from phase ϕ_0 onward, and processes in $\Pi_{<\phi_0}$ can only update $vote_p$ to ? or v, as we explain now. This ensures agreement.

First, by lines 10 and 13, it is impossible for a correct process to have two different values with the same timestamp in its pre-vote set. By lines 27-30, in phase ϕ_0, processes in $\Pi_{<\phi_0}$ can only update $vote_p$ to ?; processes in $\Pi_{=\phi_0}$ do not update neither $vote_p$, nor x_p to some value $\neq v$. By lines 10-14, in phase $\phi_0 + 1$, correct processes can only update x_p to v and can only add $(v, \phi_0 + 1)$ to *pre-vote*$_p$. Therefore in round $3(\phi_0 + 1) - 1$, correct processes can only update $vote_p$ to v, *i.e.*, only v can be decided in phase $\phi_0 + 1$. The same reasoning can be repeated for all phases after phase $\phi_0 + 1$.

Termination of $\mathcal{A}_2$: We explain intuitively termination by considering the smallest phase ϕ such that $3\phi - 2 \geq GSR$. We distinguish two cases: (i) at the beginning of round $3\phi - 2$, all correct processes have $vote_p = ?$, and (ii) at the beginning of round $3\phi - 2$ at least one correct process has $vote_p \neq ?$.

Case (i): Consider round $3\phi - 2$. A WIC round ensures that all correct processes p receive the same set $\vec{\mu}_p^{3\phi-2}$ of messages with $|\vec{\mu}_p^{3\phi-2}| \geq |\mathcal{C}|$, *i.e.*, all correct processes p set x_p to the same common value v (line 11), and add the pair $\langle v, \phi \rangle$ to *pre-vote*$_p$ (line 12). It follows that, in round $3\phi - 1$, all correct processes p set $vote_p$ to v (line 20), and all correct processes decide v in round 3ϕ (line 26).

Case (ii): This case is more complex to expose. Consider round 3ϕ, and let q be a correct process with the highest timestamp ts_q and $vote_q = v \neq ?$ at the beginning of round 3ϕ. Line 19 ensures that for any other correct process q' with $ts_{q'} = ts_q$, we have $vote_q = vote_{q'}$. Since $3\phi > GSR$, all correct processes p with $vote_p \neq v$ execute lines 27-29. Therefore, at the end of round 3ϕ all correct processes p have $x_p = v$ and $vote_p \in \{v, ?\}$, *i.e.*, all correct processes p start round $3\phi + 1 = 3(\phi + 1) - 2$ with $x_p = v$. If the condition of line 10 holds, then the most frequent pair received is $\langle v, - \rangle$, *i.e.*, $\langle v, \phi + 1 \rangle$ is added to *pre-vote*$_p$ (line 12). The condition of line 13 necessary holds at each correct process, *i.e.*, $\langle v, \phi+1\rangle$ is added to $\mathit{pre\text{-}vote}_p$ (line 14). Therefore, at the end of round $3\phi+1$, all correct processes p only have $\langle y, \phi+1\rangle$ with $y=v$ in $\mathit{pre\text{-}vote}_p$. It follows that, in round $3\phi+2$, all correct processes p set vote_p to v (line 20), and all correct processes decide v in round $3\phi+3$ (line 26).

Note that in $\mathcal{A}_2$, the set $\mathit{pre\text{-}vote}_p$ can be bounded, based on the following observation. If $\langle v, \phi\rangle \in \mathit{pre\text{-}vote}_p$ and p wants to add $\langle v, \phi'\rangle$ into its pre-vote with $\phi' > \phi$, then $\langle v, \phi\rangle$ becomes obsolete.

Proofs of $\mathcal{A}_2$:

Lemma 5.8. *Assume $n > 2t$. For all phases ϕ, and all correct processes p, there is at most one pair $\langle -, \phi\rangle$ in $\mathit{pre\text{-}vote}_p$.*

Proof. Consider round $3\phi-2$. Assume that some correct process p adds $\langle v, \phi\rangle$ to $\mathit{pre\text{-}vote}_p$ at line 14. By line 13, p received $n-t$ messages equal to $\langle v, -\rangle$. Assume for a contradiction that p has added $\langle v', \phi\rangle$, with $v' \neq v$, to $\mathit{pre\text{-}vote}_p$ at line 12. By line 11, this is only possible if p has received $n-t$ messages $\langle v', ?\rangle$. In this case, p has received $(n-t)+(n-t)$ messages in round $3\phi-2$. However, if $n > 2t$, then $(n-t)+(n-t) > n$, a contradiction. □

We define $\mathcal{P}_{agree}(3\phi-1, v)$ as the following predicate: *$\exists ts$ such that at the end of round $3\phi-1$, (i) for at least $t+1$ correct processes p we have $x_p = \mathit{vote}_p = v$ and $ts_p \geq ts$, and (ii) for other correct processes q, if $\langle v', ts'\rangle \in \mathit{pre\text{-}vote}_q$ s.t. $v' \neq v$, then $ts' \leq ts$.*

Lemma 5.9. *Assume $n > 2t$. If $\exists \phi, v$ such that $\mathcal{P}_{agree}(3\phi-1, v)$ holds, then for all $\phi' \geq \phi$, $\mathcal{P}_{agree}(3\phi'-1, v)$ also holds.*

Proof. The proof is by induction on ϕ.

Base step ($\phi' = \phi$): $\mathcal{P}_{agree}(3\phi-1, v)$ holds trivially from the assumption.

Induction step (from ϕ' to $\phi'+1$): By induction hypothesis, $\mathcal{P}_{agree}(3\phi'-1, v)$ holds. By the definition of $\mathcal{P}_{agree}(3\phi'-1, v)$, at the end of round $3\phi'-1$, (i) for at least $t+1$ correct processes p we have $x_p = \mathit{vote}_p = v$ and $ts_p \geq ts$, and (ii) for all other correct processes q, if $\langle v', ts'\rangle \in \mathit{pre\text{-}vote}_q$ s.t. $v' \neq v$, then $ts' \leq ts$. From (i) and (ii), no correct process p with $x_p = \mathit{vote}_p = v$ executes line 29 in round $3\phi'$.

Therefore $\mathcal{P}_{agree}(3\phi, v)$ holds, *i.e.*, at least $t+1$ correct processes start round $3\phi'+1$ with $x_p = \mathit{vote}_p = v$. As a consequence, in round $3\phi'+1 = 3(\phi'+1)-2$, for correct processes, (i) the condition of line 10 cannot hold and (ii) the condition of line 13 can only hold for value v. It follows that no correct process p adds $\langle v', \phi'+1\rangle$ $(v' \neq v)$ to $\mathit{pre\text{-}vote}_p$, and $\mathcal{P}_{agree}(3\phi+1, v)$ holds.

In round $3\phi'+2 = 3(\phi'+1)-1$, since no correct process sends v' and $n-t > t$ (since $n > 2t$), no correct process sets vote_p to $v' \neq v$. Therefore, $\mathcal{P}_{agree}(3\phi'+2, v)$ holds. □

Lemma 5.10. *If some correct process p decides v in round 3ϕ, then $\mathcal{P}_{agree}(3\phi-1, v)$ holds.*

Proof. From line 25, at the end of round $3\phi-1$, at least $t+1$ correct processes q have $vote_q = v$, $ts_q = \phi$, and thus $x_q = v$ (from line 20). From lines 12 and 14, in round $3\phi - 2$, no correct process adds $\langle -, x\rangle$, with $x > \phi$, to $pre\text{-}vote_p$. Therefore $\mathcal{P}_{agree}(3\phi - 1, v)$ holds. □

Proposition 5.1 (Agreement)**.** *Assume $n > 2t$. If some correct process p decides v in phase ϕ, then no correct process decides $v' \neq v$ in phase $\phi' \geq \phi$.*

Proof. From Lemma 5.10, if some correct process p decides v in phase ϕ, then $\mathcal{P}_{agree}(3\phi - 1, v)$ holds. By Lemma 5.9, $\mathcal{P}_{agree}(3\phi' - 1, v)$ holds for all $\phi' \geq \phi$. This means that, for all $\phi' \geq \phi$, at the end of round $3\phi' - 1$, for at least $t + 1$ correct processes p we have $x_p = vote_p = v$. Therefore, at least $t + 1$ correct processes p have $\langle v, -\rangle \in pre\text{-}vote_p$ in round $3\phi' - 1$. From this and Lemma 5.8, at most $n - t - 1$ processes q may have $x_q = v'$ and $\langle v', -\rangle \in pre\text{-}vote_q$ in round $3\phi' - 1$. This means that no correct process q sets $vote_q$ to v' in round $3\phi' - 1$. Therefore, in round $3\phi'$ the condition of line 25 cannot hold for $v' \neq v$. □

Lemma 5.11. *Assume $n > 3t$. In all rounds $r = 3\phi - 1$, if some correct process p sets $vote_p$ to $v \neq ?$, and some other correct process q sets $vote_q$ to $v' \neq ?$, then $v = v'$.*

Proof. Assume by contradiction that $v \neq v'$. By line 19, p receives $n - t$ messages v in round r and q receives $n - t$ messages v' in round r. From $n > 3t$, we have $(n-t)+(n-t) = 2n-2t > n+t$, or $(n-t)+(n-t) \geq n+t+1$. Therefore, $t+1$ processes have sent v to p and v' to q, *i.e.*, one correct process has sent v to p and v' to q. A contradiction with Lemma 5.8. □

Lemma 5.12. *Assume $n > 3t$. Let ϕ be the smallest phase such that round $r = 3\phi$ is after GSR. Let q be a correct process with the highest timestamp ts_q and $vote_q = v \neq ?$ at the beginning of round r. Then at the end of round r all correct processes p have $x_p = v$ and $vote_p \in \{v, ?\}$.*

Proof. At the beginning of round r, for any correct process p three cases are possible: (i) $vote_p = v$, or (ii) $vote_p = v' \neq v$, or (iii) $vote_p = ?$.

In case (i), process p does not execute line 29 in round r, but executes line 30, and sets x_p to v.

In case (ii), from Lemma 5.11, since $vote_p \neq vote_q$, $ts_p \neq ts_q$. By assumption ts_q is the highest timestamp, and so we have $ts_p < ts_q$. By line 19, at least $n - 2t$ correct processes have $\langle v, ts_q\rangle$ in their pre-vote. If $n > 3t$, then $n - 2t \geq t + 1$. Since round r is executed after GSR, all messages sent in round r are received by all correct processes. Therefore, p executes line 29 in round r, and sets $vote_p$ to ?, x_p to v.

In case (iii), process p has $ts_p = 0 < ts_q$ and for the same reason as case (ii) executes line 29 in round r, and sets x_p to v, $vote_p$ to $?$.

Therefore, at the end of round r, all correct processes p have $x_p = v$ and $vote_p \in \{v, ?\}$. □

Proposition 5.2 (Termination)**.** *Assume $n > 3t$. If round $3\phi - 2 \geq GSR$ of $\mathcal{A}_2$ is a WIC round, then $\mathcal{A}_2$ ensures termination.*

Proof. Let round $r = 3\phi - 2 \geq GSR$ be a WIC round. This implies that all correct processes receive the same set of messages in round r. Two cases are possible at the beginning of round r: (i) all correct processes p have $vote_p = ?$, or (ii) some correct process p has $vote_p \neq ?$.

In case (i), all correct processes p choose the same value v by line 11. and add $\langle v, \phi \rangle$ to *pre-vote*$_p$ in round r. By Lemma 5.8 no other pair is added to *pre-vote*$_p$ in round r. In round $r + 1 = 3\phi - 1$, all correct processes send v, receive at least $n - t$ messages v and set $vote_p = v$, $ts_p = \phi$. Finally in round $r + 2 = 3\phi$, all correct processes send $\langle v, \phi, - \rangle$, receive at least $n - t$ messages $\langle v, \phi, - \rangle$ and decide v.

In case (ii), from Lemma 5.12, all correct processes p have $x_p = v$ and $vote_p \in \{v, ?\}$ at the end of round $r + 2 = 3\phi$. All correct processes start round $r + 3 = 3\phi + 1$ with $x_p = v$. In round $3\phi + 1$, for any correct process p, if the condition of line 10 becomes true, x_p is updated to v because $n - 2t > t$. And the condition of line 13 cannot be true for $\langle v' \neq v, - \rangle$ since $n - t > t$. Therefore, no correct process p adds $\langle v' \neq v, \phi + 1 \rangle$ to *pre-vote*$_p$. By arguments similar to those of case (i), all correct processes decide v by round $r + 5 = 3(\phi + 1)$. □

To prove strong validity, we define $\mathcal{P}_{val}(3\phi - 1, v)$ as the following predicate: *at the end of round $3\phi - 1$, (i) all correct processes p have $x_p = v$, $vote_p \in \{v, ?\}$, and (ii) $\nexists v' \neq v$ s.t. $\langle v', - \rangle \in$ pre-vote$_p$.*

Lemma 5.13. *Assume $n > 3t$. If $\exists \phi, v$ such that $\mathcal{P}_{val}(3\phi - 1, v)$ holds, then for all $\phi' \geq \phi$, $\mathcal{P}_{val}(3\phi' - 1, v)$ also holds.*

Proof. The proof is by induction on ϕ.

Base step ($\phi' = \phi$): $\mathcal{P}_{val}(3\phi - 1, v)$ holds trivially from the assumption.

Induction step (from ϕ' to $\phi' + 1$): By induction hypothesis, $\mathcal{P}_{val}(3\phi' - 1, v)$ holds. By the definition of $\mathcal{P}_{val}(3\phi' - 1, v)$, at the end of round $3\phi' - 1$, all correct processes p have $x_p = v$, $vote_p \in \{v, ?\}$, and $\nexists v' \neq v$ s.t. $\langle v', - \rangle \in$ *pre-vote*$_p$. This means that in round $3\phi'$ no correct process executes line 29, and $\mathcal{P}_{val}(3\phi', v)$ holds. Therefore all correct processes p start round $3\phi' + 1 = 3(\phi' + 1) - 2$ with $x_p = v$ and $vote_p \in \{v, ?\}$.

Assume that the condition of line 10 holds at some correct process q. In this case, q has received at least $n - 2t$ messages from correct processes, and at most t messages from Byzantine processes. However, $n > 3t$ ensures $n - 2t > t$, which means that q can only add v to *pre-vote*$_q$ in line 12. Assume that the condition of line 13 holds at some correct process q. From $n > 2t$, we have $n - t > t$. Since all correct processes send $\langle v, - \rangle$, the condition of line 13 can only hold for v, which means that q can only add v to $\mathit{pre\text{-}vote}_q$ in line 14. Therefore part (ii) of $\mathcal{P}_{val}(3\phi' + 1, v)$ holds, and since *pre-vote* is not updated in round $3\phi' + 2 = 3(\phi' + 1) - 1$, part (ii) of $\mathcal{P}_{val}(3\phi' + 2, v)$ also holds.

Form this it follows that, in round $3\phi' + 2 = 3(\phi' + 1) - 1$, all correct processes send only v. From $n > 2t$, we have $n - t > t$. Therefore, the condition of line 19 can only hold for v. It follows that part (i) of $\mathcal{P}_{val}(3\phi' + 2, v)$ holds. □

Lemma 5.14. *Assume $n > 3t$. If all correct processes p have the same initial value v, then $\mathcal{P}_{val}(2, v)$ holds.*

Proof. Since all correct processes have $x_p = v$ and $\mathit{vote}_p = ?$ at the beginning of round 1, and $n - 2t > t$ no correct process p adds $\langle v', 1 \rangle$ into $\mathit{pre\text{-}vote}_p$ by lines 12 and 14. In round 2, since no correct process sends v' and $n - t > t$, no correct process votes v'. Therefore, at the end of round 2 all correct processes p have $x_p = v$, $\mathit{vote}_p \in \{v, ?\}$, and $\nexists v' \neq v$ s.t. $\langle v', 1 \rangle \in \mathit{pre\text{-}vote}_p$. This means that $\mathcal{P}_{val}(2, v)$ holds. □

Proposition 5.3 (Strong validity)**.** *Assume $n > 3t$. If all correct processes have the same initial value v, then no correct process decides $v' \neq v$.*

Proof. By Lemma 5.14, $\mathcal{P}_{val}(2, v)$ holds. By Lemma 5.13, $\mathcal{P}_{val}(3\phi - 1, v)$ holds for all $\phi \geq 1$. This means that, for all $\phi > 1$, at the end of round $3\phi - 1$, all correct processes p have $x_p = v$, and $\mathit{vote}_p \in \{v,\ ?\}$. Therefore, in round 3ϕ the condition of line 25 cannot hold for $v' \neq v$. □

Therefore we have the following theorem.

Theorem 5.3. *With Byzantine faults and $n > 3t$, if round $3\phi - 2 \geq GSR$ of $\mathcal{A}_2$ is a WIC round, then $\mathcal{A}_2$ ensures strong validity, agreement and termination.*

Proof. This follows from Proposition 5.1, 5.2, and 5.3. □

5.3.3 Summary

Table 5.1 summarizes our results. Since both the algorithms presented in the previous section and the *EIGByz* algorithm do not need any process with a specific role of leader or coordinator, the resulting consensus algorithms are decentralized. The second column shows the smallest number of processes needed for each algorithm. The third and forth columns give an upper bound on number of rounds needed for a single consensus in both best and worst cases. The best case is when the system is synchronous form the beginning,

	# processes	# rounds (best case)	# rounds (worst case)
$\mathcal{A}_1$	$5t + 1$	$t + 2$	$GSR + 2(t + 2) - 1$
$\mathcal{A}_2$	$3t + 1$	$t + 3$	$GSR + 2(t + 3) - 1$

Table 5.1: Summary of results for decentralized algorithms.

i.e., $GSR = 0$. The worst case corresponds to the case where GSR is started right after the first round of the algorithm. Both algorithms require n^2 messages per round.

5.3.4 Optimizations

We describe two possible optimizations that can be applied to our decentralized Byzantine consensus algorithm.

Early termination: The "early termination" optimization can be applied to Algorithm 5.1 (*EIGByz*). Algorithm 5.1 always requires $t + 1$ rounds, even in executions in which no process is faulty. With early termination, the number of rounds can be reduced in such cases.

Let f denote the actual number of faulty processes in a given execution. Moses and Waarts in [MW88] present an early termination version of the exponential information gathering protocol for Byzantine consensus that requires $n > 4t$ and terminates in $min\{t + 1, f + 2\}$ rounds. The idea is the following. Consider some node α in p's tree. Process p may know that a quorum (*i.e.*, $n - |\alpha| - t$) of correct children of node α store the same value. When this happens, process p can already determine the value of $newval_p(\alpha)$, and can stop at the end of the next round. The paper presents another early termination protocol with optimal resiliency ($n > 3t$) that terminates in $min\{t + 1, f + 3\}$ rounds. These two optimizations can be applied to Algorithm 5.1.

One round decision: The "one round decision" optimization is relevant to Algorithm 5.2 ($\mathcal{A}_1$). One round decision means that if all correct processes start with the same initial value, and the system is synchronous from the beginning, then correct processes decide in one single round. Algorithm 5.2 does not achieve one round decision, because the simulation of Algorithm 5.1 (*EIGByz*) appears in each phase, including phase 1. To achieve one round decision, we simply skip round 1, and start Algorithm 5.2 with round 2. If all correct processes start with the same initial value, and $GSR = 0$, then correct processes decide in one round.

The fact that our one round decision algorithm requires "only" $n > 5t$ is not in contradiction with the result in [SvR08], which establishes the lower bound $n = 7t + 1$ for one-step decision. The reason is that we assume for fast decision a partially synchronous system with $GSR = 0$, *i.e.*, the system

Algorithm 5.4 *Authenticated FloodSet* with $n > t$ (code of process p)

```
 1: Initialization:
 2:   $W_p := \{m_p\}$                          /* $m_p$ is the input value of process $p$ */

 3: Round $r$ :                                          /* $1 \leq r \leq t+1$ */
 4:   $S_p^r$:
 5:     send $\langle W_p : p \rangle$ to all processes
 6:   $T_p^r$:
 7:     for all $q$ from which the set $W_q$ is received do
 8:       for all $e \in W_q$ do
 9:         if $e$ is signed by $r$ different processes then
10:           $W_p := W_p \cup \{e\}$
11:     if $r = t+1$ then
12:       for all $q \in \Pi$ do
13:         if $(v : q : ... \in W_p)$ and $(v' : q : ... \in W_p)$ and $v \neq v'$ then
14:           remove all elements $(- : q : ...)$ from $W_p$ /* eliminate inconsistent values
              of $q$ */
15:           if $\exists v$ such that $(v : q : ...) \in W_p$ then $\vec{M}_p[q] := v$
16:           else $\vec{M}_p[q] := \bot$
```

is initially synchronous, while [SvR08] considers a system that is initially asynchronous.

5.4 Authenticated Byzantine Faults

In this section we show that decentralized Byzantine consensus is even simpler if signatures are used (a fault model called authenticated Byzantine faults, see Section 2.1.2). In this model, a faulty process who cheats about its value can be detected by the correct processes. Therefore, the *EIG-Byz* algorithm is not needed here. It can be replaced by a decentralized synchronous algorithm that uses digital signatures.

We consider here a variant of the *FloodSet* algorithm [Lyn96] (see also [DS83]) called *Authenticated FloodSet*, see Algorithm 5.4. With similar arguments as in Section 5.3, the combination of Algorithm 5.2 (or Algorithm 5.3) and Algorithm 5.4 ensures strong validity and agreement. Termination holds from Algorithm 5.4 in a partially synchronous system (after *GSR*).

In Algorithm 5.4 we denote by $v : p$ the value v signed by process p, and by $v : p_1 : p_2 : ... : p_k$ the value v signed by k processes, initially by p_1, then $v : p_1$ signed by p_2, etc. In round r processes send values signed exactly by r distinct processes, and accept only values signed exactly by r distinct processes.

At line 5 of round r, process p sends $W_p : p$, which denotes the set obtained by having p signing all elements in set W_p not yet signed by itself. In round r, a process keeps only values received that are signed by r different processes (line 9 and 10). In round $t+1$, a correct process eliminates inconsistent values, *i.e.*, two different initial values signed by the same process (line 13 and 14). At the end, every correct process constructs a vector $\vec{M}_p$ of size n, where $\vec{M}_p[q]$ is the initial value of process q (or $\perp$ if q is faulty).

5.5 Conclusion

All previously known deterministic consensus algorithms for partially synchronous systems and Byzantine faults are leader-based. However, leader-based algorithms are vulnerable to performance degradation, which occurs when the Byzantine leader sends messages slowly, but without triggering timeouts. In this chapter we have proposed a deterministic (no randomization), decentralized Byzantine consensus algorithm in a partially synchronous system. Our second algorithm is resilience-optimal (it requires $3t + 1$ processes) and signature-free (it doesn't rely on digital signatures). To the best of our knowledge this is the first Byzantine algorithm that satisfies all these characteristics. We have also presented optimizations for the Byzantine consensus algorithm, including one-round decision. Finally, a simpler decentralized consensus algorithm that uses digital signatures is proposed.

We have designed our algorithms using a new methodology which consists of extending a synchronous consensus algorithm to a partially synchronous consensus algorithm using an asynchronous algorithm. The asynchronous protocol ensures safety (*i.e.*, agreement and strong validity), while the synchronous algorithm provides liveness (*i.e.*, termination) during periods of synchrony.

We believe that our methodology is quite extensible. In fact, any algorithm that satisfies predicate $\mathcal{P}_{Int}$ in all the rounds (including synchrony and asynchrony periods), and satisfies predicate $\mathcal{P}_{Cons}$ in synchrony periods, can be extended to solve Byzantine consensus problem in a partially synchronous system. Furthermore, the same methodology that constructs a decentralized Byzantine consensus algorithm can be used to construct a new consensus algorithm for benign or timing faults. For instance, one can replace *EIGByz* algorithm by *FloodSet* algorithm and extend the *FloodSet* algorithm using *OneThirdRule* algorithm.

Chapter 6

Timing Analysis of Leader-based and Decentralized Byzantine Consensus Algorithms

The chapter compares in an analytical way two leader-based and decentralized algorithms (that is, algorithms that do not use a leader) for Byzantine consensus with strong validity. We show that for the algorithms we analyzed, in most cases, the decentralized variant of the algorithm shows a better worst-case execution time. Moreover, for the practically relevant case $t \leq 2$ (t is the maximum number of Byzantine processes), this worst-case execution time is even at least as good as the execution time of the leader-based algorithms in fault-free runs.

Publication: F. Borran and M. Hutle and A. Schiper. Timing Analysis of Leader-based and Decentralized Byzantine Consensus Algorithms. Technical Report, EPFL, 2010.

6.1 Introduction

Algorithms for solving the consensus problem can be classified into two broad categories: *leader-based* algorithms, that use the notion of a (changing) leader, and *decentralized* algorithms, where no such dedicated process is used. Most of the consensus algorithms proposed in early 80's, for both synchronous and asynchronous systems,[1] are decentralized (*e.g.*, [PSL80,LSP82, BO83, Rab83]). Later a leader (or coordinator) was introduced, in order to reduce the message complexity and/or improve the best case performance

[1] In asynchronous systems, using randomization to solve probabilistic consensus.

(*e.g.*, [DLS88, CT96, Lam98]). However, recently it has been pointed out that the leader-based PBFT Byzantine consensus algorithm [CL02], which assumes a partially synchronous system [DLS88], is vulnerable to performance degradation [ACKL08, CWA+09]. According to these two papers, a malicious leader can introduce latency into the global communication path simply by delaying the message that it has to send. Moreover, a malicious leader can manipulate the protocol timeout and slow down the system throughput without being detected. This motivated the development of decentralized Byzantine consensus algorithms for partial synchronous systems in Chapter 5. The next step, addressed here, is to compare the theoretical execution time of decentralized and leader-based consensus algorithms. We study the question analytically in the model considered in [CL02] for PBFT, namely a partially synchronous system in which the end-to-end messages transmission delay δ is unknown.

Contribution: The chapter analyzes two Byzantine consensus algorithms for strong validity, each one with a decentralized and a leader-based variant. One of these two algorithms is inspired by *Fast Byzantine Paxos* [MA06], the other by *PBFT*. Our analysis shows the superiority of the decentralized variants over the leader-based variants. First, the analysis shows that for the decentralized variants the worst case performance and the fault-free case performance overlap, which is not the case for the leader-based variants. Second, it shows that the worst case of the decentralized variant of our two algorithms is always better than the worst case of its leader-based variant. Third, for $t \leq 2$ (t is the maximum number of Byzantine processes), it shows that the worst case execution time of our decentralized variant is never worse than the execution time of the leader-based variant in fault-free runs. As future work, we plan to extend our study to consensus algorithms with weak validity, *e.g.*, Fast Byzantine Paxos and PBFT.

Roadmap: In the next section we give the system model for our analysis. Section 6.3 presents in a modular way the consensus algorithms under consideration. In Section 6.4, we give the implementation of the round model. Section 6.5 contains our main contribution, the analysis and comparison of the algorithms. We end the chapter by a discussion and a conclusion.

6.2 System Model

We consider a set Π of n processes, among which at most t can be Byzantine faulty. Processes communicate through message passing, and the system is partially synchronous [DLS88]. Instead of separate bounds on the process speeds and the transmission delay, we assume that in every run there is a bound δ on the *end-to-end transmission delay* between correct processes,

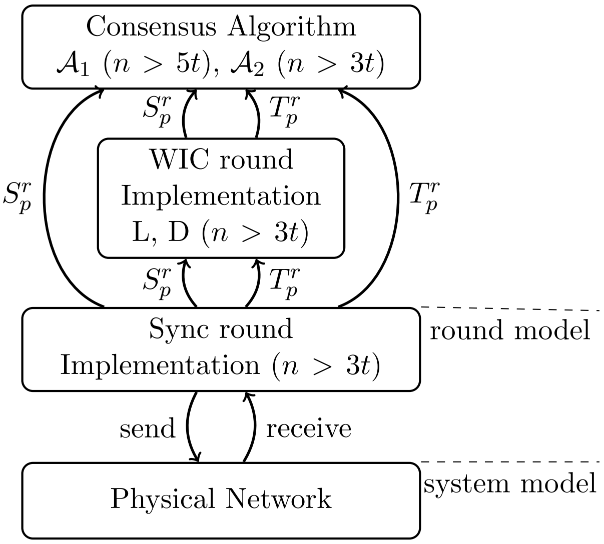


Figure 6.1: Overview of the Byzantine consensus algorithms.

that is, the time between the sending of a message and the time where this message is *actually received* (this incorporates the time for the transmission of the message and of possibly several steps until the process makes a receive step that includes this message). This is the same model considered in [CL02] for PBFT. We do not make use of digital signatures. However, the communication channels are authenticated, *i.e.*, the receiver of a message knows the identity of the sender. In addition, we assume that processes have access to a local (non-synchronized) clock; for simplicity we assume that this clock is drift-free.

As in [DLS88], we consider rounds on top of the system model defined in Section 2.3. This improves the clarity of the algorithms, makes it simpler to change implementation options, and makes the timing analysis easier to understand.

6.3 Consensus Algorithms

Consensus algorithms consist of a sequence of phases, where each phase consists of one or more rounds. Each round is composed of a sending function S and a transition function T (see Section 2.3 for details). Figure 6.1 represents an overview of the Byzantine algorithms presented in this thesis. The upper layer is the round-based consensus algorithm (one of the algorithms presented in Chapter 5, $\mathcal{A}_1$ or $\mathcal{A}_2$). As already shown in Chapter 5, our consensus algorithms require eventually a phase where all rounds are *synchronous* (defined in Section 2.3), and the first round is *consistent* (defined in Section 2.3). Eventually synchronous rounds are provided by the implementation of the round model, which is discussed in Section 6.4 (third layer from top in Figure 6.1). Ensuring eventually consistent rounds can be done in a leader-based or decentralized way, and discussed in Section 6.3 (second layer from top in Figure 6.1). By combining the two consensus algorithms with the two WIC implementations we get four algorithms that will be analyzed.

In this chapter we analyze a sequence of Byzantine consensus instances. The latency of consecutive instances of consensus is an important metric to evaluate the performance of the consensus algorithm.

6.3.1 Consensus algorithms with WIC rounds

6.3.1.A The $\mathcal{A}_1$ algorithm

According to [MHS09], the $\mathcal{A}_1$ algorithm (Algorithm 5.2 with $T = n - t$ and $E = (n + 3t)/2$) is inspired by the FaB Paxos algorithm proposed by Martin and Alvisi [MA06], expressed using rounds, including one WIC round.[2] A phase of $\mathcal{A}_1$ consists of two rounds. The algorithm is safe with $t < n/5$. For termination, the two rounds of a phase must eventually be synchronous, and the first round must be a WIC round.

6.3.1.B The $\mathcal{A}_2$ algorithm

According to [MHS09], the $\mathcal{A}_2$ algorithm (Algorithm 5.3) is inspired by the PBFT algorithm proposed by Castro and Liskov [CL02], expressed using rounds, including one WIC round.[3] A phase consists of three rounds. The algorithm is safe with $t < n/3$. For termination, the three rounds of a phase must eventually be synchronous and the first round must be a WIC round. Actually, the analysis is the same for any algorithm that requires 3 rounds per phase, with a first WIC round.

6.3.2 Implementation of a WIC round

We consider two implementations for a WIC round: one leader-based and one decentralized. The implementations are also expressed using rounds, in order to distinguish them from the "normal" rounds, we use ρ to denote these rounds. The implementation has to be understood as follows. Let r be a WIC round, *e.g.*, round $r = 2\phi - 1$ of Algorithm 5.2. The messages sent in round $r = 2\phi - 1$ are used as the input variable m_p in the WIC implementation (Algorithm 6.1). The resulting vector provided by the WIC

[2] FaB Paxos is expressed using "proposers", "acceptors" and "learners". $\mathcal{A}_1$ is expressed without these roles. Moreover, FaB Paxos solves consensus with *weak validity*, while $\mathcal{A}_1$ solves consensus with *strong validity*.

[3] PBFT solves a sequence of consensus instances with *weak validity*, while $\mathcal{A}_2$ solves consensus with *strong validity*.

Algorithm 6.1 Leader-based implementation of a WIC round with $n > 3t$ [MHS09] (code of process p)

```
Initialization:
  $\forall q \in \Pi :\ received_p[q] \leftarrow \bot$

Round $\rho = 1$:
  $S_p^\rho$:
    send $\langle m_p \rangle$ to all
  $T_p^\rho$:
    $received_p \leftarrow \vec{\mu}_p^\rho$

Round $\rho = 2$:
  $S_p^\rho$:
    send $\langle received_p \rangle$ to $coord_p$
  $T_p^\rho$:
    if $p = coord_p$ then
      for all $q \in \Pi$ do
        if $|\{q' \in \Pi : \vec{\mu}_p^\rho[q'][q] = received_p[q]\}| < 2t + 1$ then
          $received_p[q] \leftarrow \bot$

Round $\rho = 3$:
  $S_p^\rho$:
    send $\langle received_p \rangle$ to all
  $T_p^\rho$:
    for all $q \in \Pi$ do
      if $(\vec{\mu}_p^\rho[coord_p][q] \neq \bot)\ \wedge\ |\{i \in \Pi : \vec{\mu}_p^\rho[i][q] = \vec{\mu}_p^\rho[coord_p][q]\}| \geq t + 1$ then
        $\vec{M}_p[q] \leftarrow \vec{\mu}_p^\rho[coord_p][q]$
      else
        $\vec{M}_p[q] \leftarrow \bot$
```

implementation, denoted by $\vec{M}_p$ (see Algorithm 6.1) is then passed to the transition function of round r as the reception vector.

6.3.2.A Leader-based implementation

Algorithm 6.1, which appears in [MHS09], implements WIC rounds using a leader. If a correct process is the coordinator, all processes receive the same set of messages from this process in round $\rho = 3$.

In round $\rho = 2$, the coordinator compares the value received from some process p with the value indirectly received from other processes. If at least $2t + 1$ same values have been received, the coordinator keeps that value, otherwise it sets the value to $\bot$. This guarantees that if the coordinator keeps v, at least $t + 1$ correct processes have received v from p in round 1. Finally, in round $\rho = 3$ every process sends values received in round 1 or $\bot$ to all. Each process verifies whether at least $t + 1$ processes validate the value that it has received from the coordinator in round 3. Rounds 1 and 3 are thus used to verify that a faulty leader cannot forge the message from another process (integrity).

Since a WIC round can be ensured only with a correct coordinator, we

Algorithm	# rounds		# messages	
	best case	worst case	best case	worst case
$\mathcal{DA}_1$	$t+2$	$t+2$	$(t+2)n^2$	$(t+2)n^2$
$\mathcal{LA}_1$	4	$4(t+1)$	$3n^2+n$	$(3n^2+n)(t+1)$
$\mathcal{DA}_2$	$t+3$	$t+3$	$(t+3)n^2$	$(t+3)n^2$
$\mathcal{LA}_2$	5	$5(t+1)$	$4n^2+n$	$(4n^2+n)(t+1)$

Table 6.1: Results for the combination of different algorithms.

need to ensure that the coordinator is eventually correct. In Section 7.4.3 we do so by using a *rotating coordinator.* A WIC round using this leader-based implementation needs three "normal" rounds.

6.3.2.B Decentralized implementation

The decentralized (no leader) implementation of a WIC round was introduced in Chapter 5 (*EIGByz* algorithm). It is based on *Exponential Information Gathering* (EIG) algorithm for synchronous systems proposed by Pease et al. [PSL80]. Initially, process p has its input value m_p given by round $r = 2\phi - 1$ of the consensus algorithm (*e.g.*, Algorithm 5.2). Throughout the execution, processes learn about initial values of other processes. The information can be organized inside a tree. Process p maintains the tree using a set W_p. After $t+1$ rounds, badly-formatted messages in W_p are dropped, and all correct processes have the same value for W_p.

Similarly to the leader-based implementation, it requires $n > 3t$. On the other hand, a WIC round using this decentralized implementation needs $t+1$ "normal" rounds.

6.3.3 The four combinations

Combining the two WIC based algorithms, namely $\mathcal{A}_1$ and $\mathcal{A}_2$ with the two implementations of WIC rounds, namely leader-based (L) and decentralized (D), we get four algorithms, denoted by $\mathcal{LA}_1$, $\mathcal{DA}_1$, $\mathcal{LA}_2$ and $\mathcal{DA}_2$. Phases have the following lengths: four rounds for $\mathcal{LA}_1$, $t+2$ rounds for $\mathcal{DA}_1$, five rounds for $\mathcal{LA}_2$ and $t+3$ rounds for $\mathcal{DA}_2$. Table 6.1 summarizes the results for the best and worst cases, in terms of number of rounds (second column) and message complexity (third column), once the actual end-to-end transmission delay δ is reached.

6.4 Round Implementation

As already mentioned in Section 2.1, we consider a partially synchronous system with an unknown bound δ on the end-to-end transmission delay between correct processes. The main technique to find the unknown δ in the literature is using an adaptive timeout, *i.e.*, starting the first phase of an algorithm with a small timeout Γ_0 and increase it from time to time. The timeout required for an algorithm can be calculated based on the bound δ and the number of rounds needed by one phase of the algorithm. The approach proposed in the DLS model [DLS88] is increasing the timeout linearly, while recent works, *e.g.*, PBFT [CL02], increase the timeout exponentially.

The main question is when should the timeout be increased. Increasing the timeout in every phase provides a simple solution, in which all processes adapt the same timeout for a given phase. However, this is not an efficient solution, since processes might increase the timeout unnecessarily. An efficient solution is increasing the timeout when a correct process requires that. This occurs typically when a correct process is unable to terminate the algorithm with the current timeout. The problem with this solution is that different processes might increase the timeout at different points in time.

For leader-based algorithms, a related question is the relationship between leader change and timeout change. Most of the existing protocols apply both timeout and leader modifications at the same time [DLS88, CL02, MA06, KAD+07, ACKL08, CWA+09]. Our round implementation allows decoupling timeout modification and leader modification. We show that such a strategy performs better than the traditional strategies in the worst case.

6.4.1 The algorithm

Algorithm 6.2 describes the round implementation. The main idea of the algorithm is to synchronize processes into the same round (round synchronization). The algorithm requires view synchronization (eventually processes are in the same view) in addition to the round synchronization. Processes might increase the timeout not at the same round. The view number is thus used to synchronize the processes' timeout.

Each process p keeps a round number r_p and a view number v_p, initially equal to 1. While the round number corresponds also to the round number of the consensus algorithm, the view number increases only upon reconfiguration (*i.e.*, timeout modification). Thus, the leader and the timeout are functions of the view number. The leader changes whenever the view changes, based on the rotating leader paradigm (line 7). Note that the value of $coord_p$ is ignored in decentralized algorithms. The timeout does not necessarily change whenever the view changes. After line 7, a process starts the *input* & *send* part, in which it queries the input queue for new proposals (using a function *input*(), line 8), initializes new slots on the *state* vector for each new proposal (line 10), calls the send function of all active consensus instances (line 13), and sends the resulting messages (line 11). The process then sets a timeout for the current round using a deterministic function Γ based on its view number v_p (line 17), and starts the *receive* part, where it collects messages (line 22). Basically, this is done using an init/echo message

Algorithm 6.2 A round implementation for Byzantine faults with $n > 3t$ (code of process p)

```
$r_p \leftarrow 1$; $next\_r_p \leftarrow 1$                          /* round number */
$v_p \leftarrow 1$; $next\_v_p \leftarrow 1$                          /* view number */
$Rcv_p \leftarrow \emptyset$                                          /* set of received messages */
$\forall i \in \mathbb{N}: \; state_p[i] \leftarrow \bot$             /* state of instance $i$ */
$\forall i \in \mathbb{N}: \; start_p[i] \leftarrow 0$                /* starting round for instance $i$ */

while $true$ do
  $coord_p \leftarrow p_{(v_p - 1 \bmod n)+1}$
  input & send
  $I \leftarrow input()$
  for all $\langle i, v \rangle \in I$ do
    $state_p[i] \leftarrow init(v)$       /* initialization of state with initial value $v$ */
    $start_p[i] \leftarrow r_p$
  for all $i: \; state_p[i] \neq \bot$ do
    $msgs[i] \leftarrow S_p^{r_p}(state_p[i], coord_p)$
  for all $q \in \Pi$ do
    $M_q \leftarrow \{\langle i, msgs[i][q] \rangle : \; state_p[i] \neq \bot\}$
    send($\langle$START$, M_q, v_p, r_p, p\rangle$) to $q$
  receive
  $timeout_p \leftarrow current\_time + \Gamma(v_p)$
  while $next\_v_p = v_p$ and $next\_r_p = r_p$ do
    if $current\_time \geq timeout_p$ then
      send(INIT$, v_p, r_p + 1, p$) to all
    receive($M$)
    $Rcv_p \leftarrow Rcv_p \cup M$
    if exists $r$ and $t+1$ processes $q$ s.t. $\langle$INIT$, v_p, r+1, q\rangle \in Rcv_p$ then
      let $r_0$ be the largest such $r$
      if $r_0 \geq r_p$ then
        $next\_r_p \leftarrow r_0$
        send(INIT$, v_p, r_0 + 1, p$) to all
    if exists $v$ and $t+1$ processes $q$ s.t. $\langle$INIT$, v+1, -, q\rangle \in Rcv_p$ then
      let $v_0$ be the largest such $v$
      if $v_0 \geq v_p$ then
        $next\_v_p \leftarrow v_0$
        send(INIT$, v_0 + 1, r_p, p$) to all
    if exists $2t+1$ processes $q$ s.t. $\langle$INIT$, v_p, r_p + 1, q\rangle \in Rcv_p$ then
      $next\_r_p \leftarrow max\{r_p + 1, next\_r_p\}$
    if exists $2t+1$ processes $q$ s.t. $\langle$INIT$, v_p + 1, -, q\rangle \in Rcv_p$ then
      $next\_v_p \leftarrow max\{v_p + 1, next\_v_p\}$
  comp. & output
  $O \leftarrow \emptyset$
  for all $i: \; state_p[i] \neq \bot$ do
    for all $r \in [r_p, next\_r_p - 1]$ do
      $\forall q \in \Pi: \; M_r[q] \leftarrow m$ if $\exists M \; \langle$START$, M, v_p, r, q\rangle \in Rcv_p \wedge \langle i, m\rangle \in M$
                                          else $\bot$
      $state_p[i] \leftarrow T_p^r(M_r, state_p[i], coord_p)$
    if $\exists v$ s.t. $decision(state_p[i]) = v$ for the first time then
      $O \leftarrow O \cup \langle i, v\rangle$              /* $v$ is the decision of instance $i$ */
  $output(O)$
  if $v_p = next\_v_p \; \wedge \; next\_r_p \bmod \alpha = 1$ then
    if $\exists i: start_p[i] \leq next\_r_p - \alpha \; \wedge \; decision(state_p[i]) = \bot$ then
      send(INIT$, v_p + 1, next\_r_p, p$) to all
  $r_p \leftarrow next\_r_p$
  $v_p \leftarrow next\_v_p$
```

scheme based on ideas that appear already in [ST87a, DLS88, HS07]. The receive part is described later. Next, in the *comp. & output* part, the process calls the state transition function of each active instance (line 41), and outputs any new decisions (line 44) using the function *output*(). Finally, a check is done at the end of each phase, *i.e.*, only if $next\_r_p \bmod \alpha = 1$ (line 45), where α represents the number of rounds in a phase . The check may lead to request a view change, therefore, the check is skipped if $v_p \neq next\_v_p$ (the view changes anyway). The check is whether all instances started at the beginning of the phase, have decided (lines 45-46). If not, the process concludes that the current view was not successful (either the current timeout was small or the coordinator was faulty), and it expresses its intention to start the next view by sending an INIT message for view $v_p + 1$ (line 47).

The function $init(v)$ (line 10) gives the initial state for initial value v of the consensus algorithm; respectively, $decision(state)$ (line 42) gives the decision value of the current state of the consensus algorithm, or $\perp$ if the process has not yet decided.

Receive part: To prevent a Byzantine process from increasing the round number and view number unnecessarily, the algorithm uses two different type of messages, INIT messages and START messages. Process p expresses the intention to enter a new round r or new view v by sending an INIT message. For instance, when the timeout for the current round expires, the process — instead of starting immediately the next round — sends an INIT message (line 20) and waits that enough processes timeout. If process p in round r_p and view v_p receives at least $2t+1$ INIT messages for round r_p+1 (line 33), resp. view $v_p + 1$ (line 35), it advances to round $r_p + 1$, resp. to view $v_p + 1$, and sends an START message with current round and view (line 11). If the process receives $t + 1$ INIT messages for round $r + 1$ with $r \geq r_p$, it enters immediately round r (line 23), and sends an INIT message for round $r+1$. In a similar way, if the process receives $t+1$ INIT messages for view $v+1$ with $v \geq v_p$, it enters immediately view v (line 28), and sends an INIT message for view $v + 1$.

Properties of Algorithm 6.2: The correctness proofs of Algorithm 6.2 are given in Section 6.4.4. Here we give the main properties of the algorithm:

1. If one correct process starts round r (resp. view v), then there is at least one correct process that wants to start round r (resp. view v). This is because at most t processes are faulty (see Lemma 6.1).
2. If all correct processes want to start round $r+1$ (resp. view $v+1$), then all correct processes eventually start round $r + 1$ (resp. view $v + 1$). This is because $n - t \geq 2t + 1$ (see Lemma 6.2).

Strategy	A	B	C
$\Gamma(v)$	$v\Gamma_0$	$2^{v-1}\Gamma_0$	$2^{\lfloor \frac{v-1}{t+1} \rfloor}\Gamma_0$

Table 6.2: Different strategies for timeout.

3. If one correct process starts round r (resp. view v), then all correct processes eventually start round r (resp. view v). The proof is given by Lemmas 6.3-6.5.

6.4.2 Timing properties of Algorithm 6.2

Algorithm 6.2 ensures the following timing properties:

1. If process p starts round r (resp. view v) at time τ, all correct processes will start round r (resp. view v) by time $\tau + 2\delta$. Lemma 6.5 proves the property.
2. If a correct process p starts round r (view v) at time τ, it will start round $r + 1$ the latest by time $\tau + 3\delta + \Gamma(v)$. Lemma 6.6 proves the property.
3. A timeout $\Gamma(v) \geq 3\delta$ for round r (view v) ensures that if a correct process starts round r at time τ, it receives all round r messages from all correct processes before the expiration of the timeout (at time $\tau + 3\delta$). Lemma 6.7 proves the property.

6.4.3 Parameterizations of Algorithm 6.2

We now discuss different adaptive strategies for the timeout value $\Gamma(v_p)$. First we consider the approach of [DLS88]: increasing the timeout linearly (whenever the view changes). We will refer to this strategy by A. Then we consider the approach used by PBFT [CL02]: increasing the timeout exponentially (whenever the view changes). We will refer to this strategy by B. Finally, we propose another strategy, which consists of increasing the timeout exponentially every $t + 1$ views. In the context of leader-based algorithms, this strategy ensures that, if the timeout is large enough to terminate the started consensus instances, then a Byzantine leader will not be able to force correct processes to increase the timeout. We will refer to this last strategy by C. These three strategies are summarized in Table 6.2, where v represents the view number and Γ_0 denotes the initial timeout.

6.4.4 Correctness Proofs of Algorithm 6.2

In the sequel, let τ_G denote the first time that the actual end-to-end transmission delay δ is reached. All messages sent before τ_G are received the latest by time $\tau_G + \delta$. Let v_0 denote the largest view number such that no correct process has sent a Start message for view v_0 by time τ_G, but some correct process has sent a Start message for view $v_0 - 1$. Let r_0 denote the largest round number such that no correct process has sent a Start message for round r_0 by time τ_G, but some correct process has sent a Start message for round $r_0 - 1$. We prove the results related to the view number, similar results hold for round numbers:

Lemma 6.1. *Let p be a correct process that sends message $\langle \textsc{Start}, -, v, -, p\rangle$ at some time τ_0, then at least one correct process q has sent message $\langle \textsc{Init}, v, -, q\rangle$ at time $\tau \leq \tau_0$.*

Proof. Assume by contradiction that no correct process q has sent message $\langle \textsc{Init}, v, -, q\rangle$. This means that a correct process can receive at most t messages $\langle \textsc{Init}, v, -, -\rangle$ in line 28. Therefore, no correct process executes line 32, and no correct process starts view v because of line 35, which is a contradiction. □

Lemma 6.2. *Let all correct processes p send message $\langle \textsc{Init}, v, -, p\rangle$ at some time τ_0, then all correct processes p will send message $\langle \textsc{Start}, -, v, -, p\rangle$ by time $max\{\tau_0, \tau_G\} + \delta$.*

Proof. If all correct processes p send message $\langle \textsc{Init}, v, -, p\rangle$ at some time τ_0, then all correct processes are in view $v - 1$ at time τ_0 by lines 45-47. A correct process q in view $v - 1$, receives at least $n - t \geq 2t + 1$ messages $\langle \textsc{Init}, v, -, p\rangle$ by time $\tau_0 + \delta$ if $\tau_0 \geq \tau_G$, or by time $\tau_G + \delta$ if $\tau_0 < \tau_G$. From lines 35 and 36, q starts view v by time $max\{\tau_0, \tau_G\} + \delta$. □

Lemma 6.3. *Every correct process p sends message $\langle \textsc{Start}, -, v_0 - 1, -, p\rangle$ by time $\tau_G + 2\delta$.*

Proof. We assume that there is a correct process p with $v_p = v_0 - 1$ at time τ_G. This means that p has received at least $2t+1$ messages $\langle \textsc{Init}, v_0 - 1, -, -\rangle$ (line 35). Or at least $t + 1$ correct processes are in view $v_0 - 2$ and have sent a message $\langle \textsc{Init}, v_0 - 1, -, -\rangle$. These messages will be received by all correct processes the latest by time $\tau_G + \delta$. Therefore, all correct processes in view $< v_0 - 1$ receive at least $t + 1$ messages $\langle \textsc{Init}, v_0 - 1, -, -\rangle$ by time $\tau_G + \delta$, start view $v_0 - 2$ (line 31) and send a message $\langle \textsc{Init}, v_0 - 1, -, -\rangle$ (line 32). These messages are received by all correct processes by time $\tau_G + 2\delta$. Because $n - t > 2t$, all correct processes receive at least $2t + 1$ messages $\langle \textsc{Init}, v_0 - 1, -, -\rangle$ by time $\tau_G + 2\delta$ (line 35), start view $v_0 - 1$ (line 36), and send a message $\langle \textsc{Start}, -, v_0 - 1, -, -\rangle$ (line 11). □

Lemma 6.4. *Let p be the first (not necessarily unique) correct process that sends message $\langle \textsc{Start}, -, v, r, p\rangle$ with $v \geq v_0$ at some time $\tau \geq \tau_G$. Then no correct process sends message $\langle \textsc{Start}, -, v+1, -, -\rangle$ before time $\tau + TO(v)$. Moreover, no correct process sends message $\langle \textsc{Init}, v + 2, -, -\rangle$ before time $\tau + TO(v)$.*

Proof. For the Start message, assume by contradiction that process q is the first correct process that sends message $\langle \textsc{Start}, -, v+1, 1, q\rangle$ before time $\tau + TO(v)$. Process q can send this message only if it receives $2t+1$ messages $\langle \textsc{Init}, v+1, -, -\rangle$ (line 35), This means that at least $t+1$ correct processes are in view v and have sent $\langle \textsc{Init}, v+1, -, -\rangle$. In order to send $\langle \textsc{Init}, v+1, -, -\rangle$, a correct process takes at least $TO(v)$ time in view v (line 35). So message $\langle \textsc{Start}, -, v+1, -, q\rangle$ is sent by correct process q at the earliest by time $\tau + TO(v)$. A contradiction.

For the Init message, since no correct process starts view $v+1$ before time $\tau + TO(v)$, no correct process sends message $\langle \textsc{Init}, v+2, -, q\rangle$ before time $\tau + TO(v)$. □

Lemma 6.5. *Let p be the first (not necessarily unique) correct process that sends message $\langle \textsc{Start}, -, v, -, p\rangle$ with $v \geq v_0$ at some time $\tau \geq \tau_G$. Then every correct process q sends message $\langle \textsc{Start}, -, v, -, q\rangle$ by time $\tau + 2\delta$.*

Proof. Note that by the assumption, all view $v \geq v_0$ messages are sent at or after τ_G, and thus they are received by all correct processes δ time later. By Lemma 6.4, there is no message $\langle \textsc{Start}, -, v', -, -\rangle$ with $v' > v$ in the system before $\tau + TO(v)$. Process p sends message $\langle \textsc{Start}, -, v, -, p\rangle$ if it receives $2t+1$ messages $\langle \textsc{Init}, v, -, -\rangle$ (line 35). This means that at least $t+1$ correct processes are in view $v-1$ and have sent message $\langle \textsc{Init}, v, -, -\rangle$, the latest by time τ. All correct processes in view $< v$ receive at least $t+1$ messages $\langle \textsc{Init}, v, -, -\rangle$ the latest by time $\tau + \delta$, start view $v-1$ (line 31) and send $\langle \textsc{Init}, v, -, -\rangle$ (line 32) which is received at most δ time later. Because $n - t > 2t$, every correct process q receives at least $2t+1$ messages $\langle \textsc{Init}, v, -, -\rangle$ by time $\tau + 2\delta$ (line 35), start view v (line 36), and send message $\langle \textsc{Start}, -, v, -, q\rangle$ (line 11). □

Following two lemmas hold for round numbers.

Lemma 6.6. *If a correct process p sends message $\langle \textsc{Start}, -, v, r, p\rangle$ at time $\tau > \tau_G$, it will send message $\langle \textsc{Start}, -, v, r+1, p\rangle$ the latest by time $\tau + 3\delta + \Gamma(v)$.*

Proof. From Lemma 6.5 (similar result for round number), all correct processes q send message $\langle \textsc{Start}, -, v, r, q\rangle$ the latest by time $\tau+2\delta$. Then they wait for the timeout of round r which is $\Gamma(v)$ (lines 17 and 35). Therefore, by time $\tau + 2\delta + \Gamma(v)$ all correct processes timeout for round r, and send $\langle \textsc{Init}, v, r+1, q\rangle$ message to all (line 20), which takes δ time to be received by all correct processes. Finally the latest by time $\tau + 3\delta + \Gamma(v)$, process p receives $n - t \geq 2t+1$ messages $\langle \textsc{Init}, v, r+1, -\rangle$ and starts round $r+1$ (line 36). □

Lemma 6.7. *A timeout $\Gamma(v) \geq 3\delta$ for round r ensures that if a correct process p sends message $\langle \textsc{Start}, -, v, r, p\rangle$ to all at time $\tau \geq \tau_G$, it will* *receive all round messages $\langle \text{Start}, -, v, r, q\rangle$ from all correct processes q, before the expiration of the timeout (at time $\tau + 3\delta$).*

Proof. From Lemma 6.5 (similar result for round number), all correct processes q send message $\langle \text{Start}, -, v, r, q\rangle$ to all the latest by time $\tau + 2\delta$. The message of round r takes an additional δ time. Therefore a timeout of at least 3δ ensures the stated property. □

Therefore, we have the following theorem:

Theorem 6.1. *Algorithm 6.2 with $n > 3t$ ensures the the existence of round r_0 such that $\forall r \geq r_0 : \mathcal{P}_{Sync}(r)$.*

Proof. The proof holds from the previous lemmas. □

6.5 Timing Analysis

In this section we analyze the impact of the strategies A, B and C on our four consensus algorithms. We start with the analysis of the round implementation. Then we use these results to compute the execution time of k consecutive instances of consensus using the four algorithms $\mathcal{LA}_1$, $\mathcal{DA}_1$, $\mathcal{LA}_2$ and $\mathcal{DA}_2$.

First, for each strategy A, B, C, we compute the best case and worst-case execution time of k instances of repeated consensus, based on two parameters α and β: The parameter α is the one used in Algorithm 6.2. It denotes the number of rounds per phase of an algorithm, *i.e.*, the number of rounds needed to decide in the best case. Thus, α gives also the length of a view in case a process does not decide. The parameter β denotes the number of consecutive views in which a process might not decide although the timeout is already set to the correct value. This happens when a Byzantine process is the leader.

6.5.1 Best case analysis

In the best case we have $\Gamma_0 = \delta$ and there are no faults. Every round starts at the same time at all processes and takes 2δ (δ for the timeout and δ for the Init messages), and processes decide at the end of each phase (=α rounds). Therefore, the decision for k consecutive instances of consensus occurs at time $2\delta\alpha k$. Obviously, the algorithm with the smallest α (that is, the leader-based or the decentralized with $t \leq 2$) performs in this case the best.

	fault-free case		worst case	
	α	β	α	β
$\mathcal{DA}_1$	$t+2$	0	$t+2$	0
$\mathcal{LA}_1$	4	0	4	t
$\mathcal{DA}_2$	$t+3$	0	$t+3$	0
$\mathcal{LA}_2$	5	0	5	t

Table 6.3: Parameters for algorithms $\mathcal{A}_1$ and $\mathcal{A}_2$ in the worst case.

6.5.2 Worst case analysis

We compute now $\tau_X(k, \alpha, \beta)$, the worst-case execution time until the k^{th} decision when using strategy $X \in \{A, B, C\}$. In the worst case we have $\Gamma_0 \ll \delta$. Based on item 3 in Section 6.4.2 (or Lemma 6.7 in Section 6.4.4), the first decision does not occur until the round timeout is larger or equal to 3δ. We denote below with v_0 the view that corresponds to the first decision ($k = 1$).

Strategy A: With strategy A, the timeout is increased in each new view by Γ_0 until $v\Gamma_0 \geq 3\delta$, *i.e.*, until $v = \lceil 3\delta/\Gamma_0 \rceil$. Then the timeout is increased for the next β views. Therefore, we have $v_0 = \lceil 3\delta/\Gamma_0 \rceil + \beta$. To compute the time until decision, observe that a view v lasts $\Gamma(v)$ (timeout for view v) plus the time until all INIT messages are received. It is already shown that the latter takes at most 3δ (see item 2 in Section 6.4.2 and Lemma 6.6 in Section 6.4.4). Therefore we have for the worst case:

$$\begin{aligned}\tau_A(1, \alpha, \beta) &= \sum_{v=1}^{v_0} \alpha(\Gamma(v) + 3\delta) = \alpha \sum_{v=1}^{v_0} (v\Gamma_0 + 3\delta) = \alpha \left(\frac{v_0(v_0+1)}{2}\Gamma_0 + 3\delta v_0 \right) = \\ &= \alpha \left(\frac{\Gamma_0}{2}(\lceil 3\delta/\Gamma_0 \rceil + \beta)(\lceil 3\delta/\Gamma_0 \rceil + \beta + 1) + 3\delta(\lceil 3\delta/\Gamma_0 \rceil + \beta) \right) \qquad (6.1)\end{aligned}$$

and for $k > 1$,

$$\begin{aligned}\tau_A(k, \alpha, \beta) &= \tau_A(k-1, \alpha, \beta) + \alpha(v_0\Gamma_0 + 3\delta) = \\ &= \tau_A(k-1, \alpha, \beta) + \alpha(\lceil 3\delta/\Gamma_0 \rceil \Gamma_0 + \beta\Gamma_0 + 3\delta) \qquad (6.2)\end{aligned}$$

Strategy B: With strategy B, the timeout doubles in each new view until $2^{v-1}\Gamma_0 \geq 3\delta$. In other words, the timeout doubles until reaching view $v = \left\lceil \log_2 \frac{6\delta}{\Gamma_0} \right\rceil$. Including β, we have $v_0 = \left\lceil \log_2 \frac{6\delta}{\Gamma_0} \right\rceil + \beta$, and:

$$
\begin{aligned}
\tau_B(1,\alpha,\beta) &= \sum_{v=1}^{v_0} \alpha(\Gamma(v)+3\delta) = \alpha \sum_{v=1}^{v_0} (2^{v-1}\Gamma_0 + 3\delta) = \alpha\left((2^{v_0}-1)\Gamma_0 + 3\delta v_0\right) = \\
&= \alpha\left(\left(2^{\left\lceil \log_2 \frac{6\delta}{\Gamma_0}\right\rceil + \beta} - 1\right)\Gamma_0 + 3\delta\left(\left\lceil \log_2 \frac{6\delta}{\Gamma_0}\right\rceil + \beta\right)\right) = \\
&= \alpha\left(2^{\left\lceil \log_2 \frac{3\delta}{\Gamma_0}\right\rceil} 2^{\beta+1}\Gamma_0 - \Gamma_0 + 3\delta\left\lceil \log_2 \frac{3\delta}{\Gamma_0}\right\rceil + 3\delta + 3\delta\beta\right) \qquad (6.3)
\end{aligned}
$$

and for $k > 1$,

$$
\begin{aligned}
\tau_B(k,\alpha,\beta) &= \tau_B(k-1,\alpha,\beta) + \alpha(2^{v_0-1}\Gamma_0 + 3\delta) = \\
&= \tau_B(k-1,\alpha,\beta) + \alpha\left(2^{\left\lceil \log_2 \frac{3\delta}{\Gamma_0}\right\rceil} 2^{\beta}\Gamma_0 + 3\delta\right) \qquad (6.4)
\end{aligned}
$$

Strategy C: Finally, for strategy C, the timeout doubles in each new view until $2^{\frac{v-1}{t+1}}\Gamma_0 \geq 3\delta$. In other words, the timeout doubles until reaching view $v = 1 + (t+1)\left\lceil \log_2 \frac{3\delta}{\Gamma_0}\right\rceil$; then it remains the same for the next β views. Therefore we have $v_0 = (t+1)\left\lceil \log_2 \frac{3\delta}{\Gamma_0}\right\rceil + \beta + 1$, and:[4]

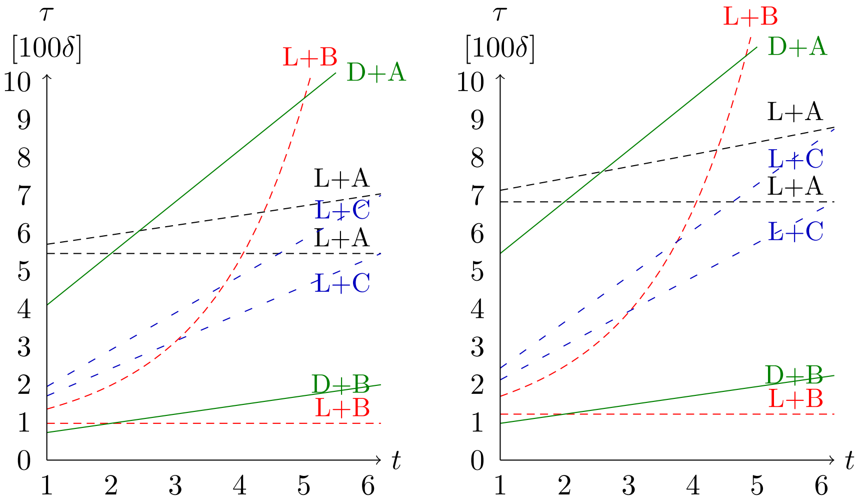
$$
\begin{aligned}
&\tau_C(1,\alpha,\beta) = \alpha\left((t+1)\sum_{l=0}^{\frac{v-1}{t+1}-1}(2^l\Gamma_0 + 3\delta) + (\beta+1)\left(2^{\frac{v-1}{t+1}}\Gamma_0 + 3\delta\right)\right) = \\
&= \alpha\left((t+1)\left(2^{\frac{v-1}{t+1}}\Gamma_0 - \Gamma_0 + 3\delta\frac{v-1}{t+1}\right) + (\beta+1)\left(2^{\frac{v-1}{t+1}}\Gamma_0 + 3\delta\right)\right) = \\
&= \alpha\left((t+1)\left(2^{\left\lceil \log_2 \frac{3\delta}{\Gamma_0}\right\rceil}\Gamma_0 - \Gamma_0 + 3\delta\left\lceil \log_2 \frac{3\delta}{\Gamma_0}\right\rceil\right) + (\beta+1)\left(2^{\left\lceil \log_2 \frac{3\delta}{\Gamma_0}\right\rceil}\Gamma_0 + 3\delta\right)\right) \\
&\qquad (6.5)
\end{aligned}
$$

and for $k > 1$,

$$
\begin{aligned}
\tau_C(k,\alpha,\beta) &= \tau_C(k-1,\alpha,\beta) + \alpha\left(2^{\frac{v-1}{t+1}}\Gamma_0 + 3\delta\right)(\beta+1) = \\
&= \tau_C(k-1,\alpha,\beta) + \alpha\left(2^{\left\lceil \log_2 \frac{3\delta}{\Gamma_0}\right\rceil}\Gamma_0 + 3\delta\right)(\beta+1) \qquad (6.6)
\end{aligned}
$$

Note that strategy C makes sense only for the leader-based algorithms.

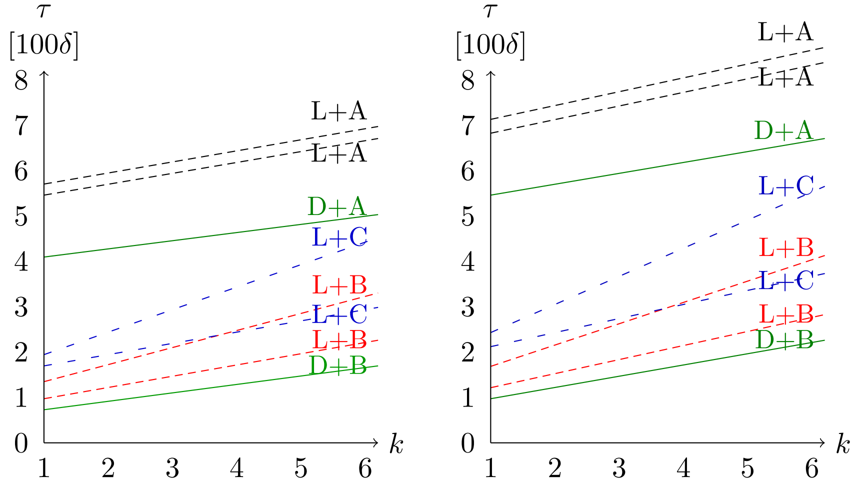
Comparison: Table 6.3 gives α and β for all algorithms we discussed. For the worst case analysis, we distinguish two cases: the *fault-free case*, which is the worst case in terms of the timing for a run without faulty process; and the general *worst-case* that gives the values for a run in which t processes are faulty.



(a) $\mathcal{A}_1$

(b) $\mathcal{A}_2$

Figure 6.2: Comparison for $k = 1$. The lower curve represents the fault-free case and the higher curve represents the worst case.



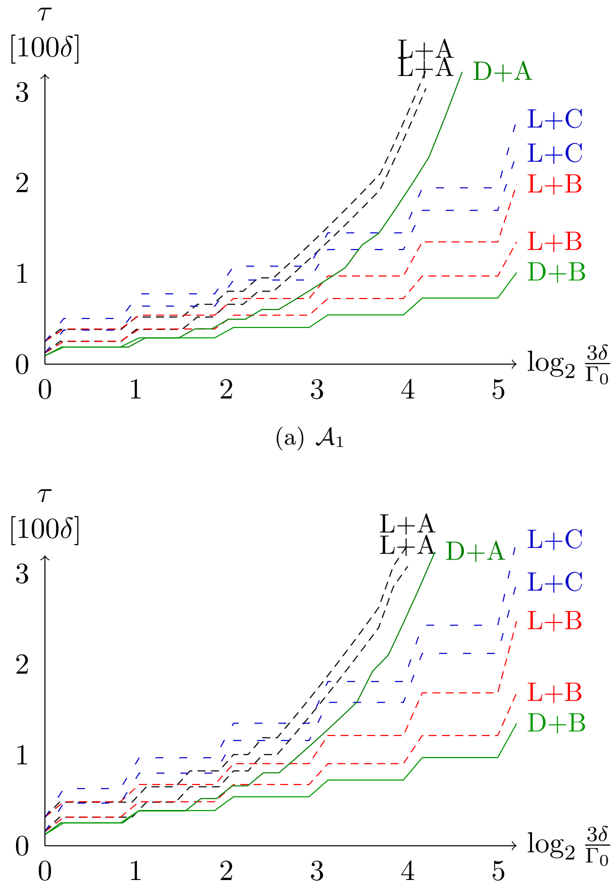
(a) $\mathcal{A}_1$

(b) $\mathcal{A}_2$

Figure 6.3: Comparison for $t = 1$. The lower curve represents the fault-free case and the higher curve represents the worst case.

We compare our results graphically in Figures 6.2-6.4. The execution time for each algorithm and strategy is a function of k, t, and the ratio δ/Γ_0. In the sequel, we fix two of these variables and vary the third.

[4] Note that from $v = 1 + (t+1)\left\lceil \log_2 \frac{3\delta}{\Gamma_0} \right\rceil$ it follows that $\frac{v-1}{t+1}$ is an integer.



(a) $\mathcal{A}_1$

(b) $\mathcal{A}_2$

Figure 6.4: Comparison of different strategies with $k = 1$ and $t = 1$. The lower curve represents the fault-free case and the higher curve represents the worst case.

We first focus on the first instance of consensus, that is, we fix $k = 1$ and assume $\delta = 10\Gamma_0$ which gives $\lceil \log_2(3\delta/\Gamma_0) \rceil = 5$, *i.e.*, the transmission delay is estimated correctly after five times doubling the timeout. The result is depicted in Figure 6.2. We first observe, as expected, that the fault-free case and the worst-case are the same for the decentralized versions. For the – in real systems relevant – cases $t < 3$, for each strategy, the decentralized algorithm decides even faster in the worst-case than the leader-based version of the same algorithm in the fault-free case. For larger t, the leader-based algorithms with strategy B are faster in the fault-free case, but less performant in the worst-case.

Next, we look how the algorithms perform for multiple instances of consensus. To this end, we depict the total time until k consecutive instances decide in Figure 6.3, for the most relevant case $t = 1$. Again we assume

$\delta = 10\Gamma_0$. Here, the decentralized algorithm is always superior to the leader-based variant using the same strategy, in the sense that even in the worst case it is faster than the corresponding algorithm in the best case. In absolute terms, the decentralized algorithms with strategy B perform the best.

Finally, we analyze the impact of the choice of Γ_0 on the execution time (Figure 6.4). This is relevant only for the first decision, *i.e.*, $k = 1$. We look at the case $t = 1$ and vary $\log_2 \frac{3\delta}{\Gamma_0}$. Again, the decentralized version is superior for each strategy. However, it can be seen that strategy A is not a good choice, neither with a decentralized nor with a leader-based algorithm, if $\log_2 \frac{3\delta}{\Gamma_0}$ is too large.

6.6 Discussion

There are two important issues that we would like to emphasize before concluding the chapter:

System model issue: The first issue is related to the round implementation. As we already mentioned, we consider a partially synchronous system where the end-to-end transmission delay is unknown. From Section 2.1.1 we know that there are two variants in this model: (i) $GST = 0$, and (ii) $GST > 0$. In the first case, there is no message loss, while in the second case there might be message loss before GST. Our round implementation (Algorithm 6.2) is correct in both system models. However, it would not be efficient in the second model, since the timeout is increased before GST, and is never decreased. For the timing analysis computed in Section 6.5 we have considered the first model. To obtain a more efficient round implementation in the second model, we suggest the following modifications (also discussed in Section 4.6 for benign faults):

- Each correct process increases its timeout according to the timeout strategy until it can solve the first instance of consensus.
- Once a first instance of consensus is solved, the process asks to reset the timeout to Γ_0 by sending a Reset message, because it believes that the actual timeout 3δ is reached.
- If a correct process receives $2t+1$ Reset messages, it resets the timeout to the initial timeout, *i.e.*, Γ_0.
- If a correct process receives $t + 1$ Reset messages, it sends a Reset message.

Using this protocol, the timeout is increased just enough to solve the consensus instance. However, each consensus instance will require the same time as the first instance. In other words, we have the following formula for the worst-case execution time until the k^{th} instance:

$$\tau_X(k, \alpha, \beta) = k \cdot \tau_X(1, \alpha, \beta).$$

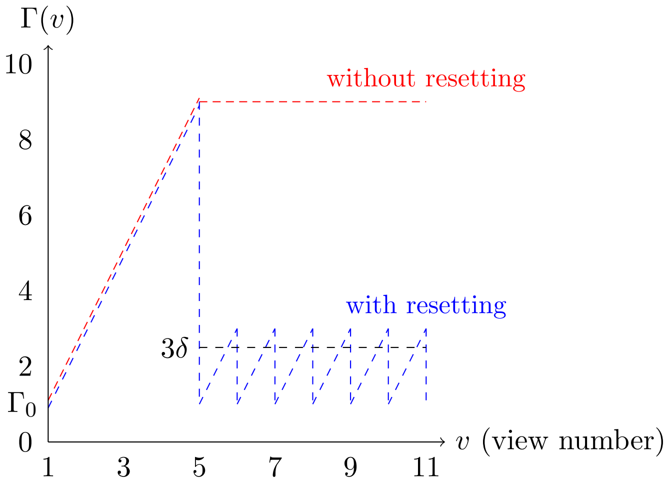


Figure 6.5: Comparing different mechanisms for timeout.

	fault-free case		worst case	
	α	β	α	β
$\mathcal{HA}_1$	4	0	$t+6$	0
$\mathcal{HA}_2$	5	0	$t+8$	0

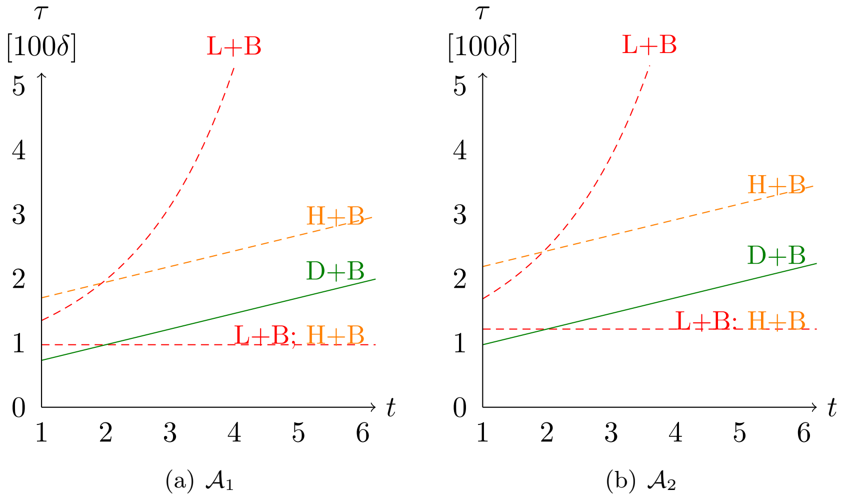
Table 6.4: Parameters for the hybrid algorithms.

where $\tau_X(1, \alpha, \beta)$ is given by the same formulas as in Section 6.5.2.

Figure 6.5 compares the previous timeout mechanism (without resetting) with the mechanism presented here (with resetting). Assuming that GST holds at view number 5, the former keeps a larger timeout comparing to the latter.

Hybrid algorithm issue: The second issue is related to the leader-based versus decentralized WIC round implementation. The leader-based version has better performance in the best case, while the decentralized version performs better in the worst case. By combining two approaches, we can obtain an algorithm that performs the best in both cases. The idea is as following, in the first phase (or view) we run the leader-based algorithm, *i.e.*, $\mathcal{LA}_1$ or $\mathcal{LA}_2$. If the first view was not successful, *i.e.*, if there is view change, then we switch to the respective decentralized algorithm, *i.e.*, $\mathcal{DA}_1$ or $\mathcal{DA}_2$.

Table 6.4 shows the parameters for the hybrid algorithm (H refers to the hybrid algorithm). Note that the hybrid algorithm is no more decentralized according to our terminology. Figure 6.6 illustrates the results of the hybrid algorithm for strategy B, and compares with the leader-based and decentralized algorithms. The hybrid algorithm is as good as the leader-based algorithm in the best case. In the worst case, the hybrid algorithm is



(a) $\mathcal{A}_1$

(b) $\mathcal{A}_2$

Figure 6.6: Comparison of hybrid algorithm for $k = 1$ and strategy B. The lower curve represents the fault-free case and the higher curve represents the worst case.

more efficient than the leader-based algorithm (for $t \geq 2$), but not as good as the decentralized algorithm.

6.7 Conclusion

We compared the leader-based and the decentralized variant of two typical consensus algorithms for Byzantine faults in an analytical way.

The results show a surprisingly clear preference for the decentralized version. While always having a better worst-cast performance, for the practically relevant cases $t \leq 2$, the decentralized variant of the algorithm is at least as good as even the fault-free case scenarios of the leader-based algorithms. But also in the best case, for $t \leq 2$, the decentralized solution is at least as good as the leader-based variant.

Part III

Wireless Networks

Chapter 7

Extending Paxos/LastVoting for Wireless Ad hoc Networks

The chapter considers the consensus problem to tolerate benign faults in wireless ad hoc networks. Most papers addressing consensus in wireless ad hoc networks adopt system models similar to those developed for wired networks. These models are focused towards node failures while ignoring link failures, and thus are poorly suited for wireless ad hoc networks. The recently proposed HO model does not have this drawback. The chapter shows that an existing algorithm and the HO model can be used for multi-hop wireless ad hoc networks, if extended with an adequate communication layer. The description of the communication layer is augmented with simulation results that validate the feasibility of our approach and provide better understanding of the behavior of the wireless environment.

Publication: F. Borran and R. Prakash and A. Schiper.
Extending Paxos/LastVoting with an Adequate Communication Layer for Wireless Ad Hoc Networks. *The 27th International Symposium on Reliable Distributed Systems (SRDS 2008)*: 227-236.

7.1 Introduction

Ad hoc networks are self-organizing wireless networks that do not rely on a preexisting infrastructure to communicate. Nodes of such networks have limited transmission range, and packets may need to traverse multiple nodes before reaching their destination. Both process and link failures are possible. Packet loss is more frequent than traditional networks due to the collisions and channel interference. In wireless networks, an algorithm with high message complexity may lead to a high number of collisions, *i.e.*, a high loss degree. In other words, it is even more important to have algorithms with low message complexity in wireless networks.

Consensus has been extensively studied in traditional networks with various system models. It is now well known that solving consensus deterministically requires some synchrony assumptions [FLP85]. One option is to assume that the (asynchronous) system eventually becomes synchronous, called partial synchrony [DLS88]; another option is to augment the (asynchronous) system with failure detectors [CT96].

Starting from this background, some papers have considered the consensus problem with benign faults in ad hoc networks. These papers essentially adopt system models similar to those developed for wired and static networks (sometimes with extensions), and these adoptations are not adequate for modeling ad hoc networks properly. Indeed, the models for wired networks are strongly biased towards node failures to the detriment of link failures. This bias has its root in the FLP paper [FLP85], which assumes process crashes and reliable links. The bias was later strengthened by the failure detector model [CT96], which also assumes process crashes and reliable links. The bias is so commonly accepted that it is easily overlooked. However, overlooking the bias results in attempts to use solutions for environments where the bias is acceptable, to environments where the bias is unacceptable. This is the case with ad hoc networks, where assuming that links are reliable is clearly inappropriate. One may argue that if reliable links are required to solve a problem then there is no work-around, and reliable links need to be implemented on top of lossy links, even if this is expensive in ad hoc networks. But this is not the case for consensus. We know that consensus can be solved in a model in which the distinction between faulty processes and faulty links completely disappears, namely the HO model [CBS09,HS07,CBS07]. This model has no bias, and is, therefore, well suited to handle transient process and link faults. Not only transient link faults (message losses) are frequent in ad hoc networks, but transient process faults can also occur (consider a wireless device that temporarily becomes unavailable due to an obstacle to signal propagation).

Having said this, the goal of the chapter is to show that an existing consensus algorithm can be used for ad hoc networks, if extended with an adequate communication layer. As suggested above, we believe that the right model for consensus in ad hoc networks is a model that handles process and link faults with the same mechanism, *e.g.*, the HO model. Several consensus algorithms have been expressed in this model, see [CBS09]. Out of these algorithms, only two of them genuinely tolerate message loss: the *OneThirdRule* (*OTR*) algorithm, and the *Paxos/LastVoting* algorithm (*LastVoting* is basically *Paxos* [Lam98] expressed in the HO model). *OTR* is certainly not adequate, because it is too costly in ad hoc networks (it requires *all-to-all* communication pattern, *i.e.*, in every step all processes send messages to all). *Paxos/LastVoting* is based on the much more economical *one-to-all* communication pattern (communication only between the coordinator and the other processes).

As in most papers about consensus in ad hoc networks, we assume that the number of nodes in the network, or at least an upper bound, is known (we comment on approaches that do not rely on this assumption in Section 7.2). Note that such an assumption is not unreasonable. Indeed, in most real wireless ad hoc networks nodes have to go through an admission before becoming operational. Also, several deployments of wireless sensors are planned deployments where a central entity makes decisions about how many and where to deploy the nodes.

Contribution: The chapter shows that an adequate communication layer can nicely handle the *one-to-all* communication pattern in multi-hop networks without any additional overhead for the routing of messages or for election of the coordinator process. The description of the communication layer is completed with simulation results that validate the feasibility of our approach and provide better understanding of the behavior of the realistic wireless environment.

Roadmap: The chapter is organized as follows. Section 7.2 presents an overview of the related work. Section 7.3 presents the consensus algorithm. Section 7.4 describes the communication layer. Simulation results are presented in Section 7.5. Section 7.6 concludes the chapter.

7.2 Related work

Several papers have addressed the consensus problem in wireless networks. One of the earliest solution to the consensus problem for a cellular network was proposed by Badache *et al.* [BHMA99]. The solution relies on a traditional fixed infrastructure of Mobile Support Stations (MSSs), and consensus is basically solved among the MSS using the Chandra-Toueg consensus protocol with the failure detector $\Diamond\mathcal{S}$ [CT96]. The MSS then propagate the decision to the mobile hosts. The solution does not address mobility.

Vollset *et al.* [VE05] propose a family of broadcast protocols to be used for solving consensus using randomization. The communication pattern is all-to-all. However, as pointed out in Section 7.1, the all-to-all communication pattern is not a good choice for multi-hop ad hoc networks. We believe that our one-to-all broadcast-convergecast algorithm is much more efficient than the general broadcast protocols proposed in [VE05].

Wu *et al.* [WCYR07], propose a consensus protocol for mobile ad hoc networks based on the failure detector $\Diamond\mathcal{P}$. Wu *et al.* recognize the problem related to the reliable link assumption, but state that complicated design changes would be needed to enable their solution to work with lossy channels. In addition to the issue of using failure detectors in ad hoc networks, the solution has another weakness. It imposes a two-layer hierarchy on the network, where k "predefined" nodes act as clusterheads. Each mobile node is associated with a clusterhead ($k < n$). The solution tolerates up to f faulty nodes, where $f < min(k, n/2)$ ($f < k$ because the solution requires one correct clusterhead). If clusterheads change during the execution, then agreeing on the clusterheads involves solving consensus which leads to circularity.

Not knowing n (called CUP, Consensus with Unknown Participants) has been considered by Cavin *et al.* [CSS04]. The paper assumes that the identity and the number of the nodes participating in the consensus are unknown, but assumes reliable channels and nodes that never crash. The notion of *participant detectors* is introduced, and the paper establishes a necessary and sufficient condition on the participant detectors for solving consensus. Later [CSS05] relaxes the requirement of non-faulty nodes, while Greve *et al.* [GT07] have extended the participant detectors of [CSS04] to include node crashes. Channels need to be reliable.

Chockler *et al.* [CDG$^+$05] developed a grid-based consensus algorithm with locally unknown participants in wireless ad hoc networks. The network is divided into a series of non-overlapping grid squares, where each grid square is assumed to be populated. Every node knows *a priori* its location in the grid. Single-hop consensus is first run for each grid square and, then, all nodes gossip the local decisions. Once a node has received a value from every grid square, it can decide by applying a deterministic function to the set of values received (which requires that every grid square provides a value). Contrary to this solution, we do not require any clustering algorithm, we do not require nodes to know their position, and we do not modify the medium access control (MAC) layer implementation. Moreover the paper makes strong synchrony assumptions (inter-node communication delay are bounded by known constants), nodes are assumed not to crash in the middle of executing a broadcast instruction, and the model does not assume node recovery after a crash. In other words a rather complex system model is considered, in contrast to our very simple model.

7.3 Consensus problem and algorithm

We consider the consensus problem for benign faults defined in Section 2.4.1. For solving consensus, we use the HO model defined in Section 2.2.

7.3.1 The Paxos/LastVoting algorithm

The *Paxos/LastVoting* algorithm [CBS09] is the most appropriate algorithm for ad hoc networks (*LastVoting* is basically *Paxos* [Lam98] expressed in the HO model, and is also close to the Chandra-Toueg $\Diamond\mathcal{S}$ [CT96] consensus algorithm): its message complexity is $O(n)$, and it tolerates rounds r in

Algorithm 7.1 *Paxos/LastVoting* algorithm (code of process p).

```
 1: Initialization:
 2:   same as Algorithm 3.1, page 30

 3: Round $r = 4\phi - 3$:
 4:   $S_p^r$:
 5:     if $Coord(p, \phi) \neq \perp$ then
 6:       send $\langle x_p, ts_p \rangle$ to $Coord(p, \phi)$
 7:   $T_p^r$:
 8:     same as Algorithm 3.1, page 30

 9: Round $r = 4\phi - 2$:
10:   same as Algorithm 3.1, page 30

11: Round $r = 4\phi - 1$:
12:   same as Algorithm 3.1, page 30

13: Round $r = 4\phi$:
14:   same as Algorithm 3.1, page 30
```

which $HO(p, r)$ is empty for all p (*i.e.*, it tolerates loss of all messages). The code (Algorithm 7.1) is similar to Algorithm 3.1 given in Chapter 3 as *LastVoting in four rounds*, page 30, with one exception in line 6 (process p sends message to $Coord(p, \phi)$ only if the latter is non-$\perp$). The reason for this modification is the following. Contrary to the predicate implementations given in Section 3.5 and 3.6, the communication layer given in Section 7.4 can provide no coordinator for some processes at the beginning of the algorithm (see line 4 of Algorithm 7.2). For every process p, the communication layer provides the coordinator of p in phase ϕ, denoted by $Coord(p, \phi)$, and the messages received from the set $HO(p, r)$. From here on we call the algorithm simply *LastVoting*. The liveness predicate of the *LastVoting* algorithm is the same as in Chapter 3 page 29, *i.e.*, $\mathcal{P}_{lv4}$.

7.4 Communication layer for LastVoting

The communication layer's role is to ensure the predicate $\mathcal{P}_{lv4}$, which includes the election of a coordinator.

7.4.1 System model

Wireless network: We consider an asynchronous multi-hop wireless network consisting of set of n nodes.[1] We use the terms *node* and *process* interchangeably. Each node in the network has a single wireless transceiver through which it can communicate with other nodes. The maximum distance at which a node's transmission can be successfully received may be

[1] Actually n needs only to be an upper bound of the number of nodes.

less than the upper bound on the communication range. Moreover, this distance may change from one transmission to the next. This is different from the unit-disk graph model, and a more realistic representation of wireless propagation characteristics.

Unreliable links and unpredictable delays: When employing MAC layer broadcast, the transmitter does not necessarily know the identities of all nodes within its communication range. Nor does the transmitter know the subset of nodes that successfully received the message. Broadcast communication satisfies the basic *integrity* and *no-duplication* properties guaranteeing that every received message was previously broadcast, and each message is received at most once. However, it is inherently *unreliable*: the receivers do not send any acknowledgment, and the sender does not make any retry attempts to increase the likelihood of message delivery to neighbors. Though MAC layer unicast is described as being reliable (uses acknowledgments), there is no guarantee that a data frame will be forwarded to the intended neighbor. So, we assume that the wireless links are *unreliable* and the message communication delay is *unpredictable*: our algorithm doesn't require any protocol like TCP, unlike [WCYR07].

Node crashes: In addition to link failures, nodes can crash. Faults can be transient or permanent, but a majority of nodes must remain connected despite permanent faults. Note that this does not prevent nodes from being temporarily disconnected.

Good period: *LastVoting* is always safe. To ensure liveness, we must restrict the asynchrony of the system. We assume that, from time to time, unknown to the processes, the system experiences good periods, during which messages are reliably transmitted with the end-to-end (multi-hop) transmission delay bounded by a known constant δ.[2] Note that this is not in contradiction with our previous assumption about unreliable links and unpredictable delays. This is required to overcome FLP impossibility result. The notion of good period is a more realistic system assumption than partially synchronous systems, inspired from [DLS88] and already used in [HS07].

7.4.2 Architecture

Figure 7.1 shows the overall view of our architecture. The uppermost layer corresponds to *LastVoting* (Algorithm 7.1). *LastVoting* contains two functions S_p^r and T_p^r that are called by the layer beneath, namely Algorithm 7.2:[3]

[2] It would be easy to adapt the algorithm to an unknown δ value [DLS88], *e.g.*, using adaptive timeout.

[3] Actually *LastVoting* does not send the messages in lines 6, 18, 26, 33; it simply defines which messages should be sent to which destinations.

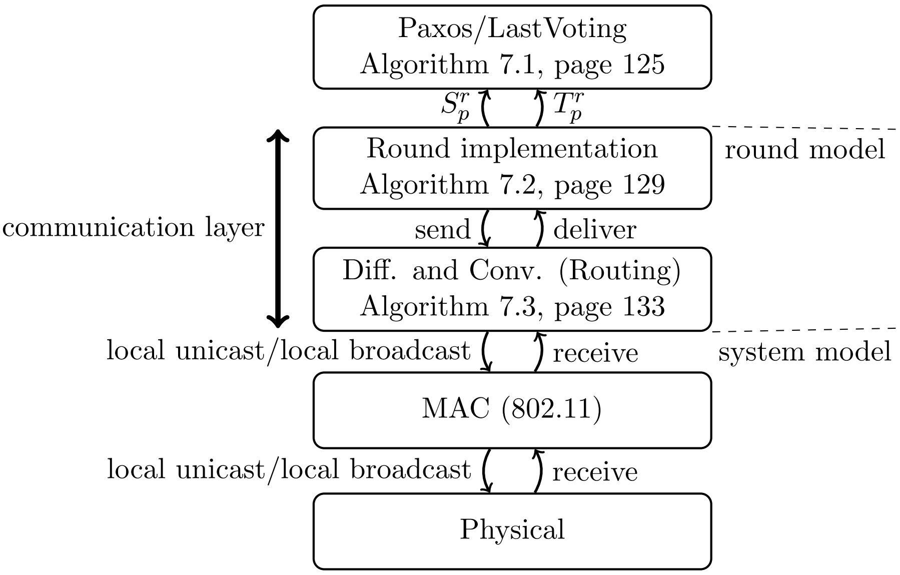


Figure 7.1: Architecture of the *Paxos/LastVoting* protocol.

- The sending step S_p^r of Algorithm 7.1 is a function $S_p^r(s_p, coord_p)$ that takes as input the round number r, the state s_p, the coordinator $coord_p$, and returns the set msg of message(s) to be sent, together with their destination(s) dst (see Algorithm 7.2, line 15).
- The state transition step T_p^r of Algorithm 7.1 is a function $T_p^r(msgs, s_p, coord_p)$ that takes as input the round number r, the set of messages received ($msgs$), the state s_p, the coordinator $coord_p$, and returns the new state ns_p (see Algorithm 7.2, line 32).

Algorithm 7.2 uses Algorithm 7.3 as a simple and best-effort broadcast and convergecast algorithm on top of the MAC sub-layer, which typically uses a CSMA/CA-based protocol like IEEE 802.11. For sending a message, Algorithm 7.2 calls the *send* function of Algorithm 7.3. Upon reception of a message by Algorithm 7.3, the *deliver* function of Algorithm 7.2 is called. Both MAC layer broadcasts and unicasts are used by Algorithm 7.3: when a message has to be locally broadcast, the MAC layer broadcast primitive is used.

7.4.3 Algorithm 7.2: the upper communication layer

For every process p, Algorithm 7.2 has two main roles:

- Elect the coordinator (to be used as a parameter of the S_p^r function).
- For every round r, construct the set of messages received by p (to be used as a parameter of the T_p^r function).

Before discussing these two issues, some general explanations are needed. First, note that Algorithm 7.2 handles the process state s_p (line 3), the phase number ϕ_p (line 8) and the round number r_p (line 14). Second, Algorithm 7.2 relies on Algorithm 7.3 for sending (and receiving) messages (*e.g.*, line 17): the routing implemented by Algorithm 7.3 is optimized to drop unnecessary messages. Third, Algorithm 7.2 is designed to ensure fast phase synchronization once a good period has started. Phase synchronization is needed, since when a good period starts, processes can be in different phases (and different rounds). Fast phase synchronization means that processes quickly join the same phase, in order to allow processes to decide. This is done as follows: Each process attaches its current phase number ϕ_p and round number r_p to the messages it sends (*e.g.*, line 17). Whenever a process receives a message from some phase $\phi > \phi_p$, it jumps to the first round of that phase (line 31, 12).

Coordinator election: Each process has a priority (*e.g.*, the process identity, line 5), and the process that believes to have the highest priority for some phase ϕ becomes the coordinator for that phase. To be more efficient, the coordinator is restricted to a predefined set $Contender \subset \Pi$.[4] Initially, every process $p \in Contender$ considers itself as a coordinator (line 4).

At the beginning of each phase ϕ, every process p that considers itself to be coordinator, sends its identity and priority to all (line 11). This is the only message that Algorithm 7.2 sends in addition to the messages of Algorithm 7.1. Each process $p \in \Pi$ that receives a message from phase $\phi \geq \phi_p$ from some process q with higher priority (line 23, 28), updates its coordinator to q and its priority to q's priority.

After the beginning of a good period, let τ be the time at which the first process starts some new phase ϕ_0 (other processes are in earlier phases: with smaller phase numbers). Then at time $\tau + 2\delta$ there is a unique coordinator c for all phases $\geq \phi_0$. However, a unique coordinator c at time $\tau + 2\delta$ is not enough to ensure termination in phase ϕ_0: multiple coordinators between τ and $\tau+2\delta$ can prevent a decision in phase ϕ_0. So phase ϕ_0+1 is started after 2δ in case c is still in round $4\phi_0 - 3$ (line 37); c is the unique coordinator for the remainder of the good period.

Message reception: For every round r, Algorithm 7.2 constructs the set of messages received by process p (to be used as a parameter of the T_p^r function). This is done differently whether $p \in Contender$ or $p \notin Contender$. If $p \notin Contender$, then p does not use a timer; if $p \in Contender$ then p uses a timer.

[4] The *Contender* set must be large enough to ensure that all its members are not crashed at the same time.

Algorithm 7.2 Upper communication layer: Coordinator election and message reception (code of process p)

```
Initialization:
   $msgs_p \leftarrow \emptyset$                                   /* set of messages received */
   $s_p \leftarrow init_p$                                          /* state of process $p$ */
   $coord_p \leftarrow p$ for $p \in Contender$; otherwise $\perp$
   $priority_p \leftarrow p$'s identity for $p \in Contender$; otherwise 0
   startPhase (1)

function startPhase ($\phi$)
   $\phi_p \leftarrow \phi$                                              /* phase number */
   if $p \in Contender$ then $timer_p \leftarrow 0$
   if $p = coord_p$ then
      send ($\langle \phi_p, -, p, priority_p, - \rangle, \Pi$)      /* calls function send of Algorithm 7.3;
      message used to elect coordinator; $\Pi$ is the destination set */
   startRound ($4\phi_p - 3$)

function startRound ($r$)
   $r_p \leftarrow r$                                                /* round number */
   $\langle msg, dst \rangle \leftarrow S_p^{r_p}(s_p, coord_p)$          /* calls function S of Algorithm 7.1 */
   if $msg \neq null$ then
      send ($\langle \phi_p, r_p, p, priority_p, msg \rangle, dst$)  /* calls function send of Algorithm 7.3 */

function deliver ($\langle \phi, r, q, priority_q, m \rangle$)            /* called by Algorithm 7.3 */
   if $\phi < \phi_p$ or $r < r_p$ then
      ignore message
   else
      $msgs_p \leftarrow msgs_p \cup \{\langle \phi, r, q, priority_q, m \rangle\}$
      if $\phi = \phi_p$ and $priority_q > priority_p$ then
         $coord_p \leftarrow q$; $priority_p \leftarrow priority_q$;
      if $\phi > \phi_p$ then
         $coord_p \leftarrow p$ for $p \in Contender$; otherwise $\perp$
         $priority_p \leftarrow p$'s identity for $p \in Contender$; otherwise 0
         if $priority_q > priority_p$ then
            $coord_p \leftarrow q$; $priority_p \leftarrow priority_q$;
         forall $r' \in [r_p, r)$ do $s_p \leftarrow T_p^{r'}(\{\langle m, q \rangle | \langle \phi_p, r', q, -, m \rangle \in msgs_p\}, s_p, coord_p)$
         /* calls function T for intermediate rounds */
         startPhase ($\phi$)
      $ns_p \leftarrow T_p^r(\{\langle m, q \rangle | \langle \phi_p, r, q, -, m \rangle \in msgs_p\}, s_p, coord_p)$     /* calls function T of
      Algorithm 7.1 */
      if $ns_p \neq s_p$ then        /* new state of $p$ is different from its current state */
         $s_p \leftarrow ns_p$; startRound ($r + 1$);

upon $timer_p > 5\delta$ do                        /* timeout for current phase expires */
   $coord_p \leftarrow p$; $priority_p \leftarrow p$'s identity; startPhase ($\phi_p + 1$);

upon $timer_p > 2\delta$ do            /* start new phase if no progress as coordinator */
   if $p = coord_p$ and $r_p < 4\phi_p - 2$ then startPhase ($\phi_p + 1$)

upon decide for phase $\phi_p$ do
   if $p = coord_p$ then startPhase ($\phi_p + 1$)
```

Case 1: $p \notin$ Contender. In this case p remains in the current round r_p of phase ϕ_p until (1) it receives a message from a larger phase (line 25) or (2) p has received "enough" messages in round r (lines 32 to 34). Note that Algorithm 7.2 does not know what "enough" means. "Enough" is defined by Algorithm 7.1: in rounds $4\phi - 3$ and $4\phi - 1$ "enough" is more than $n/2$; in rounds $4\phi - 2$ and 4ϕ "enough" is 1. The solution is for Algorithm 7.2 to call the T_p^r function whenever a new message is received (line 32): if not enough messages have been received, the T_p^r function does not modify the state (line 33) and p remains in the same round (in order to wait for more messages).

Case 2: $p \in$ Contender. In addition to behaving like an ordinary process (Case 1), p uses a timer, which is reset at the beginning of each phase ϕ_p (line 9). In a good period a round does not take more than δ. So, in addition to the behavior explained under Case 1, p remains in phase ϕ_p until (1) 2δ time units have elapsed (duration of coordinator election round and round $4\phi - 3$) and p is still in round $4\phi_p - 3$ (line 37), or (2) 5δ time units have elapsed (duration of coordinator election round and rounds $4\phi - 3$ to 4ϕ) and p is still in phase ϕ_p (line 35).

Optimizations: Algorithm 7.2 includes two optimizations. The first one is useful when several instances of consensus are running one after the other (*e.g.*, atomic broadcast). When a decision occurs in phase ϕ, the coordinator starts immediately phase $\phi + 1$ (line 39) without waiting the timeout for phase ϕ. The second optimization avoids unnecessary coordinator changes. Once some process p is considered to be the coordinator by a majority, it remains the coordinator as long as its messages reach a majority of processes: process $q \in$ *Contender* that considers p as its coordinator ($priority_q < priority_p$) does not change its coordinator unless its timer expires (line 35). Finally, another optimization – not shown in Algorithm 7.2 but considered in our simulations – is the following: the coordinator, on starting a new phase (lines 38 and 40), does not need to send an additional message to all (line 11), because there is a unique coordinator. This additional message has to be sent when there is no unique coordinator: either timer has expired (line 36) or a message from higher phase is received (line 31).

7.4.4 Proofs

Theorem 7.1. *Algorithm 7.2 implements the predicate $\mathcal{P}_{lv4}$ in a good period of minimal length 13δ.*[5]

Proof. The proof is based on the following Lemmas. □

[5] δ is the end-to-end multi-hop transmission delay.

Lemma 7.1. *Let phase ϕ_0 be the largest phase when a good period starts at time τ_G. Then, there is some process that starts phase $\phi_0 + 1$ at latest by time $\tau_G + 5\delta$.*

Proof. According to the code of Algorithm 7.2, all contenders start a timer per phase (line 9). According to the definition of contender set, there is at least one process (that is up) in contender set. This process times out for phase ϕ_0 at latest by time $\tau_G + 5\delta$ (line 35), and starts phase $\phi_0 + 1$ (line 36) at latest by time $\tau_G + 5\delta$. □

Lemma 7.2. *Let p be the first (not necessarily unique) process that starts phase ϕ_0 at time $\tau > \tau_G$. Then, process p belongs to the Contender set.*

Proof. From Lemma 7.1 process p exists. According to the Algorithm 7.2, a process starts phase ϕ_0 for following reasons, either: (i) it receives a message from another process for phase ϕ_0 (line 31), or (ii) it ends phase $\phi_0 - 1$ by deciding (line 40), or (iii) its timer for phase $\phi_0 - 1$ expires (line 36), or (iv) after 2δ, the coordinator does not receive from a majority set (line 38). The first case is not possible, since p is the first process that starts phase ϕ_0. In the second case, we have $p = coord_p$ which implies $p \in Contender$ by definition. For the two last cases, since p has a timer (line 9) it is already a contender. □

Lemma 7.3. *Let p be the first (not necessarily unique) process that starts phase ϕ_0 at time $\tau > \tau_G$. Then, all processes start phase ϕ_0 at latest by time $\tau + \delta$.*

Proof. From Lemma 7.2 we have $p \in Contender$. According to the Algorithm 7.2, process p starts phase ϕ_0 by sending a message to all (line 11). Since we are in good period, this message will be received by all processes at latest by $\tau + \delta$. All processes that receive this message start phase ϕ_0. If some process at phase $\phi_0 - 1$ times out, just before receiving this message, it starts phase ϕ_0 on its own before $\tau + \delta$. Thus, all processes start phase ϕ_0 at latest by time $\tau + \delta$. □

Lemma 7.4. *Let p be the first (not necessarily unique) process that starts phase ϕ_0 at time $\tau > \tau_G$. Then, all processes have the same coordinator by time $\tau + 2\delta$.*

Proof. According to the Lemma 7.3, all processes start phase ϕ_0 at latest by time $\tau + \delta$. Assume there is some other process $q \in Contender$ such that $priority_q > priority_p$. Process q starts phase ϕ_0 at time t (line 31), $\tau < t < \tau + \delta$, considering itself as coordinator (line 26), and sends its first message for phase ϕ_0 to all (line 11). This message will also be received by all processes at latest by time $t + \delta < \tau + 2\delta$. All processes change their coordinator to q (line 24) before $\tau + 2\delta$. □

Lemma 7.5. *Let p be the unique coordinator with highest priority that starts phase ϕ_0 at time $\tau > \tau_G$. Then, Algorithm 7.2 ensures $\mathcal{P}_{lv4}$ by time $\tau + 5\delta$.*

Proof. Process p starts phase ϕ_0 by sending its message to all (line 11). All processes receive this message by time $\tau + \delta$ (Lemma 7.3) and start round $4\phi_0 - 3$ (line 12). Since p is the unique coordinator of phase ϕ_0, no other process executes line 11. Since p is the process with highest priority, all processes accept p as coordinator in phase ϕ_0 (line 29). Since we are in good period, a round does not take more than δ. Algorithm 7.1 requires four rounds (4δ). In total at latest by time $\tau + 5\delta$ the predicate $\mathcal{P}_{lv4}(\phi_0)$ is satisfied. □

Lemma 7.6. *Let p be the first (not necessarily unique) process that starts phase ϕ_0 at time $\tau > \tau_G$. Let $c \neq p$ be the coordinator of phase ϕ_0 with highest priority that receives only from a minority of processes in round $4\phi - 1$. Then, process c starts phase $\phi_0 + 1$ at latest by time $\tau + 3\delta$.*

Proof. From Lemma 7.3, process c starts phase ϕ_0 at latest by time $\tau + \delta$. From Lemma 7.4, process c becomes the unique coordinator of phase ϕ_0 at latest by time $\tau + 2\delta$. From the code of Algorithm 7.2, process c, 2δ after starting phase ϕ_0, finds out that it has not received from a majority of processes (line 37). So, it starts phase $\phi_0 + 1$ at latest by time $\tau + 3\delta$ (line 38). □

Lemma 7.7. *Let p be the first (not necessarily unique) process that starts phase ϕ_0 at time $\tau > \tau_G$. Then, the predicate $\mathcal{P}_{lv4}(\phi_0)$ is satisfied by time $\tau + 8\delta$.*

Proof. Two cases are possible: either p is the process with highest priority or not. In the first case, from Lemma 7.5, the predicate is satisfied by time $\tau + 5\delta$. In the second case, from Lemma 7.4, there is a unique coordinator, c, by time $\tau + 2\delta$. Process c starts phase $\phi_0 + 1$ at latest by time $\tau + 3\delta$ according to Lemma 7.6. In phase $\phi_0 + 1$, process c is the unique coordinator and again according to the Lemma 7.5 the predicate is satisfied by time $\tau + 8\delta$. □

Analysis: From Lemma 7.1, we have seen that at most 5δ after τ_G a new phase is started. From Lemma 7.7, we need 8δ to satisfy the predicate $\mathcal{P}_{lv4}$. In total, we need a good period of minimal length 13δ to ensure the predicate.

7.4.5 The lower communication layer: broadcast and convergecast

Algorithm 7.2 invokes Algorithm 7.3 (lower communication layer) when it sends a message in lines 11 and 17. Depending on *dst*, Algorithm 7.3 uses

Algorithm 7.3 Lower communication layer: broadcast and convergecast algorithm (code of process p)

```
1:  Initialization:
2:     $parent_p \in \Pi \cup \{\text{NULL}\}$, initially NULL
3:     $level_p \in \mathbb{N}$, initially 0
4:     $priority_p$ refers below to the variable $priority_p$ of Algorithm 7.2

5:  function send $(m, dst)$                          /* called by Algorithm 7.2 */
6:     if $dst = \Pi$ then
7:        $parent_p := p$; $level_p := 1$;
8:        locally broadcast $\langle MESSAGE, p, level_p, m\rangle$
9:     else
10:       unicast $\langle RESPONSE, q, level_p, m\rangle$ to $parent_p$

11: upon receive $\langle MESSAGE, root, l, m\rangle$ from node $q$ with $priority_q$ for the first time
    do
12:    deliver $(m)$                                   /* calls Algorithm 7.2 */
13:    if $priority_q > priority_p$ then
14:       $parent_p := q$; $level_p := l + 1$;
15:    if $priority_q \geq priority_p$ then
16:       locally broadcast $\langle MESSAGE, root, level_p, m\rangle$

17: upon receive $\langle RESPONSE, root, l, m\rangle$ for the first time do
18:    if $p = root$ then deliver $(m)$                /* calls Algorithm 7.2 */
19:    else unicast $\langle RESPONSE, root, level_p, m\rangle$ to $parent_p$
```

diffusion or convergecast in lines 8 and 10: diffusion is used for a message sent by a coordinator (*one-to-all*), while convergecast is used for messages sent to the coordinator (*all-to-one*). Diffusion messages are identified by the tag MESSAGE (*e.g.*, line 8), while convergecast messages are identified by the tag RESPONSE (*e.g.*, line 10). During diffusion, Algorithm 7.3 delivers the message that is received for the first time (line 12) to Algorithm 7.2. During convergecast, the message is delivered only if it reaches its destination (line 18). Algorithm 7.3 also contributes to an efficient election of the coordinator by discarding messages from contenders that can no more become coordinator.

Diffusion: As all participating nodes are not within communication range of each other, it is not possible for a node to directly communicate with all others. However, a network-wide message broadcast can be implemented through diffusion. The message source (a coordinator) will broadcast the message locally at the MAC layer (line 8). When node p receives a message from some node q for the first time (line 11), and $priority_q > priority_p$ (q wins against p), then p becomes a child of q (line 14) and p broadcasts the message at the MAC layer (line 16). When a node receives copies of the same message later, it ignores them. *As a result, an efficient tree rooted at a coordinator is formed.*

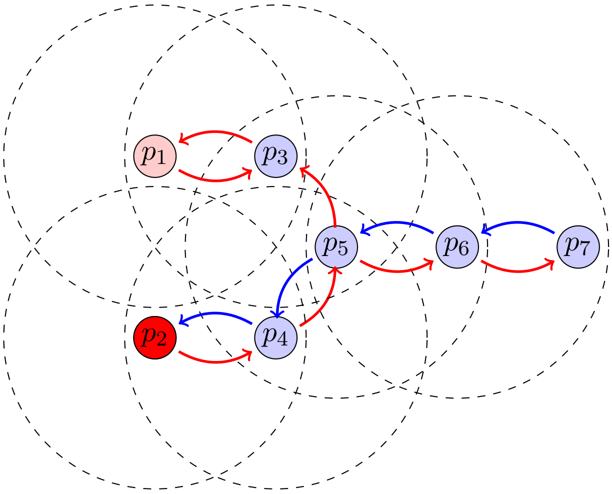


Figure 7.2: Broadcast (red arrows) *vs.* convergecast (blue arrows). $p_1, p_2 \in Contender$ ($p_2 > p_1$). p_2 becomes the coordinator.

Convergecast: The tree constructed during diffusion is used by convergecast, to transport responses to the coordinator, the root of the tree. As a node does not know the identities of all its children, it is not possible for the node to determine when it has received responses from all of them. Therefore, each node sends its response to its parent as soon as the node joins the tree. Subsequently, whenever the node receives a response from any child it forwards the received response to its parent. Figure 7.2 shows an example of broadcast and convergecast protocol in a multi-hop network. During diffusion (tag MESSAGE), since p_2's priority is higher than p_1's, if p_5 receives the message form p_2 before p_1, it ignores p_1's message. Otherwise, it diffuses both, but p_4 becomes its parent and p_2 its grand parent. During convergecast (tag RESPONSE), only path from p_7 to p_2 is followed.

Gradient-based convergecast: If any node on the path from node p to the root of the tree (*i.e.*, to the coordinator) is down, or any link on this path is lossy, p's message may not reach the root. Gradient-based convergecast can increase the probability of responses reaching the root. During diffusion, as a node joins the tree, it sets its level to be one greater than its parent's level (line 14). The root is always at level one (line 7). During convergecast nodes listen to transmissions in the promiscuous mode. If they receive a message from a neighboring node at a higher level they retransmit the message (using MAC layer broadcast). Thus, messages travel from higher level to lower level, with no cyclic forwarding, ultimately reaching the root. Even if the path from the root to a node breaks down after the node has joined the tree, it may be possible for the node's response to reach the root along other gradient-based paths, if such paths exist. This can be done as follows:

1. In line 10, instead of sending the RESPONSE to the parent, locally broadcast the RESPONSE.
2. In line 19, first determine if $l > level_p$. If so, locally broadcast the RESPONSE.

7.5 Simulation

We used the JiST/SWANS v1.0.6 [BHvR$^+$04,Bar04] wireless network simulator. We consider a $m \times m$ square grid with nodes placed at each intersection as illustrated in Figure 7.3. The grid-based placement is used instead of the random uniform placement only for manageability reasons. For instance, using this placement we can select exactly which nodes belong to the *Contender* set. Communication between two nodes p_1 and p_2 occurs in an ad hoc manner using unicast/broadcast as defined in the IEEE 802.11b standard [Gro97]. The data rate of the wireless channel is 1 Mbps. All nodes have the same transmission range (150 m). We modify the network area to vary network density and network diameter. Nodes are stationary, except for one case in which we measure the impact of mobility (see Section 7.5.2.E). We measure the impact of location and number of contenders in Section 7.5.2.C. Each contender starts the algorithm randomly between 0 and 10 milliseconds after simulation start time. The simulation lasts for 100 seconds. Every consensus packet is around 32 bytes.

Note that the IEEE 802.11b MAC layer specification uses CSMA/CA and enforces RTS/CTS/ACK control frames for unicast communication only. Collision control for broadcast is limited to basic collision avoidance carrier sensing, and broadcast is therefore prone to packet collisions. A straightforward approach to reduce collisions is to have nodes wait for a small random amount of time (jitter) before rebroadcasting.

Given the consensus algorithm in Section 7.3.1 and based on broadcast and convergecast protocol (Algorithm 7.3), we are interested in analyzing whether the required liveness condition is provided by Algorithm 7.2 and 7.3 in wireless ad hoc networks. The network that we consider is quite dense to avoid partitioning as much as possible, and quite noisy (due to frequent collisions, node interference, and background traffic explained later) to simulate bad periods.

7.5.1 Metrics

In order to evaluate the performance of the *LastVoting* consensus algorithm, several instances of consensus are run one after the other. Each process starts a new instance of consensus with a new proposition. A new consensus

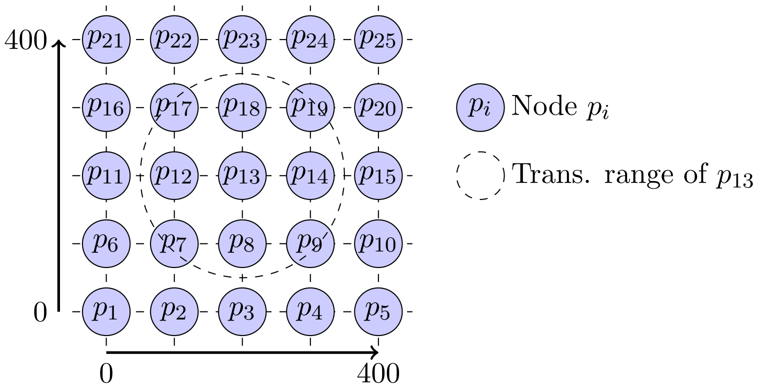

400
0
0
400
p_{21} p_{22} p_{23} p_{24} p_{25}
p_{16} p_{17} p_{18} p_{19} p_{20}
p_{11} p_{12} p_{13} p_{14} p_{15}
p_6 p_7 p_8 p_9 p_{10}
p_1 p_2 p_3 p_4 p_5
p_i Node p_i
Trans. range of p_{13}

Figure 7.3: Square grid of size 5 × 5 in network area 400 × 400 m^2.

instance is started as soon as the decision for current instance is reached or a message from a later instance is received. In the latter case, the previous decisions can be communicated through piggy-backing.

We have defined two (independent) metrics: *consensus latency* and *consensus throughput*. Consensus latency is expressed in terms of average number of phases per consensus from initialization to first decision. Consensus throughput represents the number instances run successfully during the simulation time (100 seconds): the time for one consensus is simply $100/throughput$.

7.5.2 Results

We evaluate the performance of our consensus algorithm in both single and multi-hop networks. In these scenarios no process crashes and no packet is explicitly dropped: the only source of failure is the collisions and node interferences.[6] However, to observe the performance of our algorithm in realistic situations, we added a background traffic to the system: every second, each node sends a packet (with the same size as a consensus packet) to a random destination. We have noticed that increasing background traffic only reduces the throughput of our algorithm slightly (additional graphs can be found in the Appendix A). All results of simulations are averaged over 30 independent runs. The vertical bars in the graph represent 95% confidence interval for the mean.

First, we ran a calibration test to examine the behavior of the simulator and our routing algorithm to tune the amount of the jitter. Figure 7.4 (top) shows once a single message has been broadcast, the duration for which the wireless channel remains busy (henceforth, referred to as channel occupancy

[6] Considering only message loss does not make consensus easier to solve: consensus is impossible to solve in a synchronous system with lossy links [SW89]. To solve consensus, message loss must be restricted.

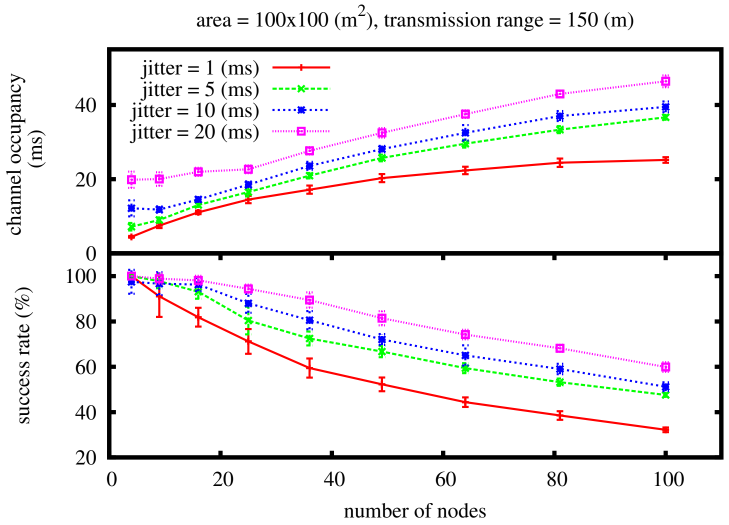


Figure 7.4: Impact of network density and jitter.

duration). Note that the same message forwarding algorithm is employed by each node: on receiving a message for the first time, a node rebroadcasts the message after a random time between 0 and jitter. So, the wireless channel becomes idle either when the message is received by everyone or when the message is completely lost. For instance, for 100 nodes within range of each other, with jitter = 10 ms, channel occupancy is 40 ms. This gives us 80 ms for round-trip time, or 200 ms for one phase of our consensus implementation (5×40 ms). Figure 7.4 (bottom) shows the percentage of nodes that receive the broadcast message. It seems that the value of the jitter is optimal around 10 ms. With 10 ms, at least a majority of processes have received the message (*LastVoting* requires a majority) and there is almost the same channel occupancy as 5 ms. For the rest of simulations we fix jitter to 10 ms.

7.5.2.A Single-hop scenarios

First, we consider a single-hop network in which all nodes are in communication range of each other. The network area is $100 \times 100\ m^2$. We gradually increased the network density. Only a single node, for example p_1, belongs to the *Contender* set. We measured the average number of phases per consensus in networks with different node densities (from 4 nodes to 100 nodes) while varying the timeout. The value of timeout refers to 5δ used in Algorithm 7.2. The ideal value in our scenario is 1 phase per consensus. However, this value can increase in the presence of packet loss.

Figure 7.5 (top) shows how the number of phases varies with timeout. Logarithmic scales are used in x-axis to better visualize a large range of

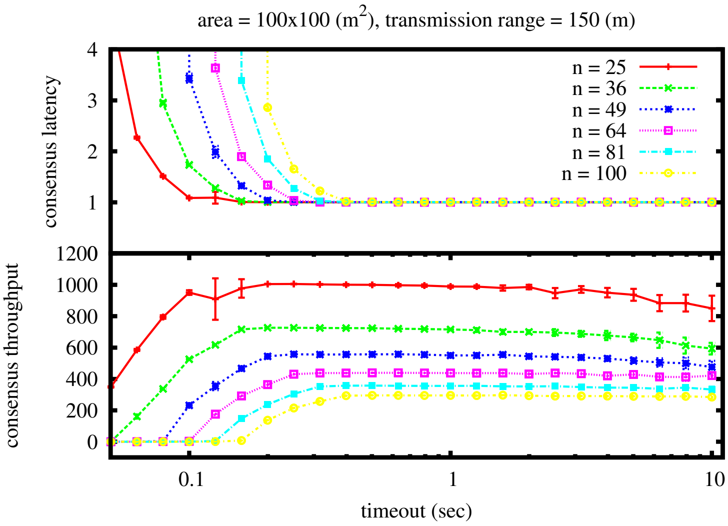


Figure 7.5: Impact of timeout in single-hop networks.

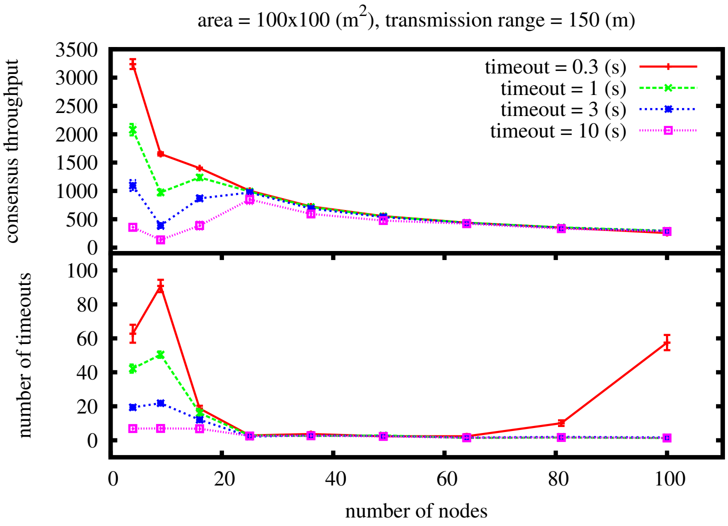


Figure 7.6: Impact of density in single-hop networks.

timeout and emphasize the small timeouts. Beyond a certain value of timeout, the number of phases to terminate consensus remains almost constant (1 phase) as density of the deployment increases. Figure 7.5 (bottom) shows how consensus throughput varies with timeout for several network densities. Note that the results we have obtained in this simulation based on the timeouts confirm our previous results on channel occupancy (*e.g.*, for $n = 64$ the best throughput is for $timeout \approx 200$ (ms), which is greater than 5×30 (ms) taken from Figure 7.4 (top)).

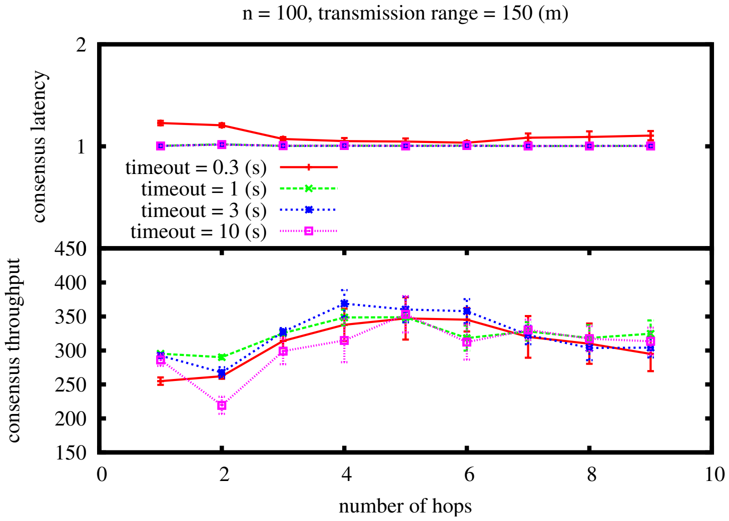


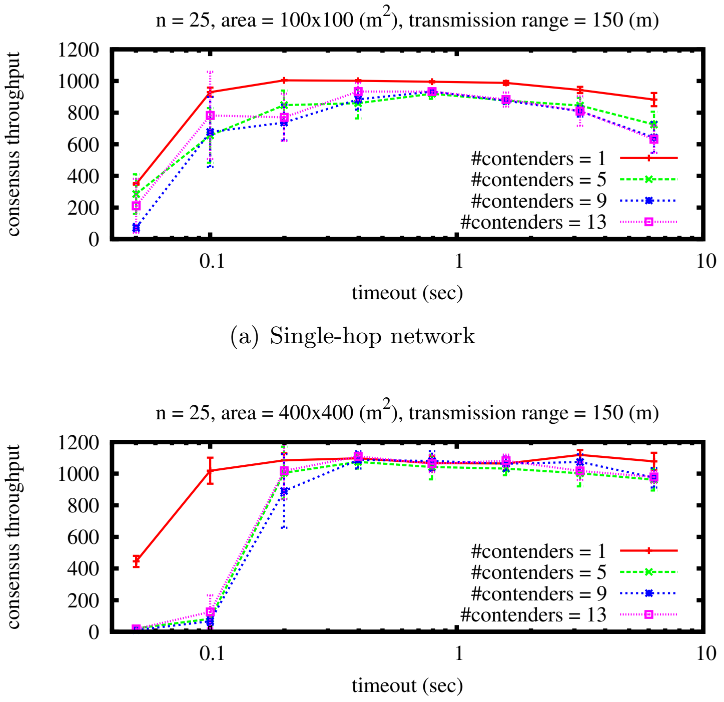
Figure 7.7: Impact of network diameter in multi-hop networks.

Figure 7.6 (top) is just another representation of Figure 7.5 (bottom) to better visualize the impact of network density. In general, by increasing density (number of nodes), the throughput of our algorithm decreases, independent of the phase timeout value. This is because of message losses due to increased collisions. The graph shows that there is an optimal value for density. After around 25 nodes, the throughput always goes down. So the algorithm performs less efficiently in the presence of more than 25 nodes per 10000 m^2 (single-hop). Although with small number of nodes the throughput is high, the number of timeouts that occur is also high (see Figure 7.6 (bottom)). For instance, for $n = 4$ the algorithm allows only one message loss while for $n = 100$, 49 losses are allowed in a round (majority set). This explains why for small number of nodes, increasing the timeout reduces the performance, see Figure 7.6 (top).

7.5.2.B Multi-hop scenarios

We consider 100 nodes distributed in a 10×10 square grid. The transmission range for each node is fixed to 150 m. To obtain multi-hop scenarios, we varied the network area from 100×100 m^2 (single-hop) to 900×900 m^2 (9-hops), and we chose p_1 as the coordinator ($p1$ is located at the lower left corner of the grid).

Figure 7.7 (top) shows the scalability of our algorithm in multi-hop networks. By increasing the network area for 100 nodes, on one hand we increase the number of hops and on the other hand we decrease the density and, therefore, the probability of message collisions. Figure 7.7 (bottom) shows the trade-off between number of hops and network density. From



(a) Single-hop network

(b) Multi-hop network

Figure 7.8: Impact of contenders.

one-hop to four-hops, we decrease the density, so the performance is improved. From six-hops on, since the message must traverse more hops the performance is slightly decreased. So, 100 nodes perform better in five-hops. This gives approximately 20 nodes per hop. This is almost the same conclusion that we had from single-hop scenarios.

7.5.2.C Impact of location and number of contenders

We varied the position of the contender (coordinator) from bottom-left corner to the center.[7] We run a Kruskal-Wallis non-paired data test [Bou06] (generalized Wilcoxon Rank Sum test) to determine if the position of the contender influences consensus throughput (null hypothesis: position of the contender does not influence consensus throughput). The test accepts the null hypothesis with p-value 0.9699. The conclusion is that the throughput of our consensus algorithm is independent of the contender's position. This seems reasonable in single-hop networks. In multi-hop networks, when the contender moves from bottom-left corner to the center of square grid, the

[7] This is enough to explore other possibilities because of the symmetry of the square grid.

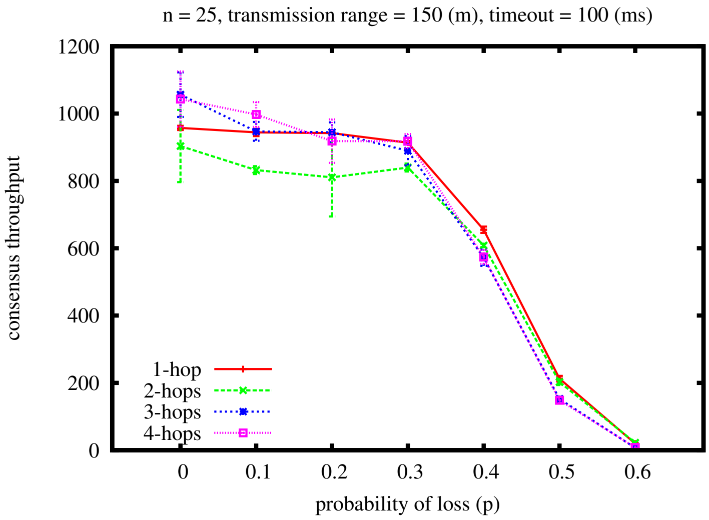


Figure 7.9: Impact of message loss.

number of hops from the contender to the farthest node is reduced while the number of collision is augmented (in center there is 4 times more collision than in corner). So in multi-hop networks, reduced number of hops is compensated by increased number of collisions.

In Figure 7.8, we increased the number of contenders in a network of 25 nodes from 1% to 50%. The figure confirms that for large enough timeouts, the number of contenders does not have an important impact on the consensus throughput. In fact, once the process with highest priority is elected as the coordinator, it remains the same as long as a majority of its messages are not lost.

To better understand how the crash of the coordinator influences our algorithm, we ran a simulation in which the contender with the highest priority crashes and recovers with frequency $1/t$. We noticed that for $t \gg timeout$ (which is reasonable assumption) the consensus throughput is almost the same as the case in which the contender never crashes (the corresponding graphs are in the Appendix A).

7.5.2.D Impact of message loss

We now consider scenarios in which a node on receiving a message discards it with probability p (uniform distribution). This simulates the loss of the message during its passage through the network. There is one coordinator located at the lower left corner of the grid. Figure 7.9 shows the sensitivity of our algorithm to p and confirms the ability of our algorithm to tolerate almost 50% message loss. Until 30% message loss the performance is almost the same.

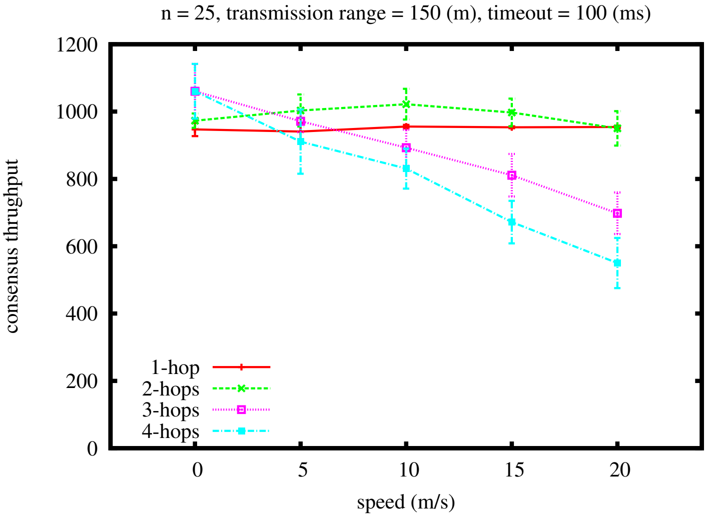


Figure 7.10: Impact of mobility.

7.5.2.E Impact of mobility

Finally, we measured the impact of mobility on consensus throughput. We used the random waypoint model with a fixed speed and zero pause time. In this model, nodes select an arbitrary destination in the field and move directly towards it at constant speed. When they reach the destination, they pick a new destination and so on. Figure 7.10 shows the behavior of our algorithm with node speed. The coordinator is located at the lower left corner of the grid at the beginning of the simulation, and then moves. Note that when the network diameter is 2 (network area is 200 × 200), a majority of nodes (13 nodes) are in communication range with the coordinator, which explains why there is no difference between 1-hop and 2-hop scenarios.

7.6 Conclusion

The *Paxos/LastVoting* algorithm extended with an adequate communication layer can potentially solve the consensus problem in wireless mobile networks. *Paxos/LastVoting* is safe by design, but a communication predicate is required to ensure the termination of consensus. We have proposed an appropriate implementation that satisfies the required communication predicate in good periods. We have validated our implementation by running simulations in multi-hop wireless networks. The results of simulations validate the existence of the good periods and confirm that our approach is applicable for realistic networks.

We could not compare our results with Chockler's paper [CDG$^+$05] since they do not provide the time unit in their figures. The results in Vollset's paper [VE05] are far from being efficient (they require around 100 seconds in average for one instance of consensus). Finally, the performance evaluation in Wu's paper [WCYR07] is of limited utility since they do not use a realistic MAC layer in their simulations. Although the results of this chapter are limited to the simulations, we believe that this approach is applicable in real systems.

Chapter 8

Conclusion

This chapter assesses the research performed in the context of the thesis and presents directions for future research.

8.1 Thesis Assessment

The thesis has investigated the round model abstraction, to represent consensus algorithms for benign and Byzantine faults in a concise and modular way. The round model allowed us to separate the consensus algorithms from the round implementations, propose different round implementations for a single consensus algorithm, improve existing round implementations, extend round implementation for wireless ad hoc networks and provide quantitative analysis of different algorithms. In the context of Byzantine faults, the round model allowed us to better understand existing protocols, express them in a simple and modular way, obtain simplified proofs, discover new protocols, and finally perform precise timing analysis to compare different algorithms. In the following, we highlight the main contribution of the thesis in more details.

Quantitative analysis of consensus algorithms In the context of the HO model and benign faults, we have shown different implementations for the HO predicates required by the consensus algorithms. We have also derived the time complexity (and message complexity) for implementations in a system that alternates between good and bad periods. The time complexity is the duration of the good period for one instance of consensus. For each algorithm two expressions are computed: one for the case of a short good period (that includes an initialization period plus the duration for a regular phase), and another for long good period (for which the initialization period can be ignored). Our results show that the OTR consensus algorithm has always the best time complexity, but requires a two-thirds majority of processes to be up in a good period. Among the algorithms that require only a simple majority of up processes, their time complexity is very close in the case of a short good period. In the case of a long good period, the difference becomes relevant. For example in the case $\Phi \ll \Delta$, which is typically the case in a WAN, LV-4 with synchronization by a coordinator has a time complexity 50% higher than LV-3 with piggybacking. However, the message complexity of the latter is higher ($n^2 + 2n$ vs. $4n$).

Swift algorithms for repeated consensus We have analyzed efficiency of algorithms in two models for solving consensus: the round-based model (which can be implemented on top of a partially synchronous system), and the asynchronous system augmented with failure detectors. Efficiency refers here to *swiftness*, a new notion that captures the fact that an algorithm, once the system is stabilized, progresses at the speed of the messages. Our new round-based implementation combines the advantages of failure detector solutions (swiftness) and round-based model (lossy links). This weak link assumption makes round-based algorithm easy to adapt to the crash-recovery model with stable storage [HS07]. We have illustrated the new round-based implementation on a specific consensus algorithm (OTR). This does not mean that the new solution is limited to OTR. It applies to any consensus algorithm expressed in the round model, in particular to the *LastVoting* algorithm [CBS09], a round-based variant of Paxos [Lam98]. Furthermore, we show another swift round implementation for benign faults that can be extended for Byzantine faults.

Decentralized Byzantine consensus algorithms All previously known deterministic consensus algorithms for partially synchronous systems with Byzantine faults are leader-based. However, leader-based algorithms are vulnerable to performance degradation, which occurs when the Byzantine leader sends messages slowly, but without triggering timeouts. Our results confirm the existence of a deterministic decentralized Byzantine consensus algorithm in a partially synchronous system that is resilient-optimal and signature-free. We started from the observation that decentralized consensus algorithms exist for the synchronous system, both for benign faults (*e.g.*, the *FloodSet* algorithm [Lyn96]) and for Byzantine faults (*e.g.*, the algorithm based on interactive consistency [PSL80]). However, these algorithms violate agreement if executed during the asynchronous period of a partially synchronous system. Therefore we combined one of these algorithms with a second algorithm that ensures agreement in an asynchronous system. One of our decentralized Byzantine consensus algorithms requires $5t + 1$ processes, and $t + 2$ rounds per consensus instance; the other one requires $3t + 1$ processes, and $t + 3$ rounds per consensus instance during periods of synchrony.

Timing analysis of leader-based and decentralized Byzantine consensus algorithms We compared the leader-based and the decentralized variant of two typical round-based consensus algorithms for Byzantine faults in an analytical way. The results show a surprisingly clear preference for the decentralized version in the worst case. Moreover, the decentralized variant of the algorithm is at least as good as the leader-based algorithms in both fault-free and best case scenarios for the practically relevant cases $t \leq 2$. We also showed a hybrid algorithm that is as good as the leader-based algorithm in the fault-free case, and has a better performance than the leader-based algorithm (comparable to the decentralized algorithm) in the worst case.

Extending Paxos/LastVoting for wireless ad hoc networks We have shown how to extend the *Paxos/LastVoting* algorithm with an adequate communication layer to solve the consensus problem in wireless mobile ad hoc networks. *Paxos/LastVoting* is safe by design, but a communication predicate is required to ensure the termination of consensus. We have proposed an appropriate implementation that employs a best-effort broadcast/convergecast protocol to satisfy the required communication predicate in good periods. We have validated our implementation by running simulations in single hop and multi-hop wireless networks. The results of simulations validate the existence of the good periods and confirm that our approach is applicable for realistic networks. Although the results are limited to the simulations, we believe that this approach is applicable in real systems.

8.2 Future Research Directions

The thesis can be extended in the following directions:

Swift algorithms for Byzantine faults In the thesis we have shown two different approaches to construct swift algorithms to tolerate benign faults (see Chapter 4): one is using the *Alive* set and terminating rounds earlier, the other is using an adaptive timeout mechanism. The first approach seems not to be applicable for Byzantine faults, while the second approach is used in Chapter 6. One possible direction for future research is to investigate other possibilities to render a round based algorithm swift with benign and Byzantine faults. One important advantage of the round-based approach comparing to other ad hoc approaches is that the core of the consensus algorithm remains the same for both non-swift and swift algorithms, and the same round implementation can be used by different consensus algorithms.

General transformations for consensus algorithms In Chapter 5 we have shown the main idea of our algorithm to derive a decentralized algorithm. The idea is not limited to decentralized algorithms nor to Byzantine faults.

We believe that our methodology is quite extensible. In fact, any algorithm that satisfies predicate $\mathcal{P}_{Int}$ in all the rounds (including synchrony and asynchrony periods), and satisfies predicate $\mathcal{P}_{Cons}$ in synchrony periods, can be extended to solve consensus problem in a partially synchronous system. A future work is to investigate other synchronous Byzantine consensus algorithms, *e.g.*, *phase queen* and *phase king* algorithms [BG89, BGP89], to see whether they satisfy the above mentioned properties and are extensible or not. Furthermore, the same methodology used to construct a decentralized Byzantine consensus algorithm can be used to construct a new consensus algorithm for benign or timing faults. For instance, one can replace *EIGByz* algorithm by *FloodSet* algorithm and extend the *FloodSet* algorithm using *OneThirdRule* algorithm.

Compare decentralized and leader-based consensus algorithms experimentally We have compared two decentralized and leader-based Byzantine consensus algorithms in an analytical way in Chapter 6. It may be interesting to compare the algorithms experimentally to better understand the trade-off between the execution time and the message complexity. The message complexity of our decentralized algorithm is quadratic in number of processes and its message size is exponential in number of faults, while the leader-based algorithm requires one round with linear message complexity the others being quadratic.

Extending decentralized Byzantine consensus algorithm for state machine replication The Byzantine consensus algorithms presented in Chapter 5 ensure the strong validity property of consensus. Existing protocols for state machine replication, *e.g.*, PBFT, satisfy only weak validity property. An interesting future work would be to extend our Byzantine consensus algorithms to atomic broadcast. Having this, a fair comparison between PBFT and decentralized PBFT would be possible. In fact the work can be extended to implement generic broadcast [PS02] protocols.

Extend and simulate a Byzantine consensus algorithm for MANETs Message complexity is an important factor for wireless ad hoc networks. However, it seems that any Byzantine consensus algorithm that satisfies strong validity property requires a quadratic number of messages even in the fault-free case, unless using digital signatures. If the claim is true, it would be interesting to compare the trade-off between quadratic message complexity and digital signatures for the wireless ad hoc networks.

Experiment the proposed consensus algorithms for MANETs using a real testbed Conducting a real experiment for wireless ad hoc networks is a complex and time-consuming task. FRANC [CSS03] is a lightweight Java framework to implement and deploy real ad hoc network applications. Future work could explore the performance of the proposed consensus algorithms using FRANC in multi-hop and mobile wireless ad hoc networks.

Bibliography

[ACKL08] Y. Amir, B. Coan, J. Kirsch, and J. Lane. Byzantine replication under attack. In *Dependable Systems and Networks (DSN'08)*, pages 197–206, 2008.

[ACKM06] I. Abraham, G. Chockler, I. Keidar, and D. Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. *Distributed Computing*, 18(5):387–408, 2006.

[ACT98] M. K. Aguilera, W. Chen, and S. Toueg. Failure detection and consensus in the crash-recovery model. In *Proceedings of the 12th International Conference on Distributed Computing (DISC'98)*, pages 231–245. Springer-Verlag, 1998.

[ADGFT03] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *Proceedings of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*. ACM Press, 2003.

[AEMGG$^+$05] M. Abd-El-Malek, G. R. Ganger, G. R. Goodson, M. K. Reiter, and J. J. Wylie. Fault-scalable byzantine fault-tolerant services. *SIGOPS Oper. Syst. Rev.*, 39(5):59–74, 2005.

[AGGT08] D. Alistarh, S. Gilbert, R. Guerraoui, and C. Travers. How to solve consensus in the smallest window of synchrony. In *Proceedings of the 22nd International Conference on Distributed Computing (DISC'08)*, pages 32–46, 2008.

[AW04] H. Attiya and J. Welch. *Distributed Computing.* John Wiley & Sons, 2nd edition, 2004.

[Bar04] R. Barr. *An efficient, unifying approach to simulation using virtual machines.* PhD thesis, Cornell University, Ithaca, NY, May 2004.

[Baz00] R. A. Bazzi. Synchronous byzantine quorum systems. *Distrib. Comput.*, 13(1):45–52, 2000.

[BCBG$^+$07] M. Biely, B. Charron-Bost, A. Gaillard, M. Hutle, A. Schiper, and J. Widder. Tolerating corrupted communication. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC'07)*. ACM Press, 2007.

[BG89] P. Berman and J. A. Garay. Asymptotically optimal distributed consensus (extended abstract). In *16th International Colloquium on Automata, Languages and Programming (ICALP'89)*, pages 80–94, London, UK, 1989. Springer-Verlag.

[BGP89] P. Berman, J. Garay, and K. Perry. Towards optimal distributed consensus. In *30th Annual Symposium on Foundations of Computer Science (FOCS'89)*, pages 410–415, Los Alamitos, CA, USA, 1989. IEEE Computer Society.

[BH09] M. Biely and M. Hutle. Consensus when all processes may be byzantine for some time. In *Proceedings of the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'09)*, pages 120–132, Berlin, Heidelberg, 2009. Springer-Verlag.

[BHMA99] N. Badache, M. Hurfin, R. Macedo, and P. Adp. Solving the consensus problem in a mobile environment. In *Proceedings of the IEEE International Performance, Computing, and Communications Conference (IPCCC99). Phoenix/Scottsdale*, pages 29–35. IEEE Press, 1999.

[BHR00] R. Baldoni, J.-M. Hélary, and M. Raynal. From crash fault-tolerance to arbitrary-fault tolerance: Towards a modular approach. In *Dependable Systems and Networks (DSN'00)*, pages 273–282, 2000.

[BHSS09] F. Borran, M. Hutle, N. Santos, and A. Schiper. Quantitative analysis of consensus algorithms. Technical report, EPFL, 2009. Submitted to TDSC.

[BHvR$^+$04] R. Barr, Z. J. Hass, R. van Renesse, K. Tamtoro, B. S. Viglietta, C. Lin, M. Fong, and E. Cheung. JiST/SWANS, 2004. http://jist.ece.cornell.edu.

[BO83] M. Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC'83)*, pages 27–29. ACM Press, 1983.

[Bou06] J.-Y. L. Boudec. Methods, practice and theory for the performance evaluation of computer and communication systems, 2006. Lecture Notes, EPFL, Switzerland.

[Bra84] G. Bracha. An asynchronous [(n - 1)/3]-resilient consensus protocol. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, pages 154–162, New York, NY, USA, 1984. ACM.

[Bra87] G. Bracha. Asynchronous byzantine agreement protocols. *Inf. Comput.*, 75(2):130–143, 1987.

[BT83] G. Bracha and S. Toueg. Resilient consensus protocols. In *Proceedings of the 2nd Annual ACM Symposium on Principles of Distributed Computing (PODC'83)*, pages 12–26, New York, NY, USA, 1983. ACM.

[BW09] M. Biely and J. Widder. Optimal message-driven implementations of omega with mute processes. *ACM Trans. Auton. Adapt. Syst.*, 4(1):1–22, 2009.

[CBPE10] B. Charron-Bost, F. Pedone, and S. A. (Eds.). *Replication: Theory and Practice.* Springer, 2010.

[CBS06] B. Charron-Bost and A. Schiper. Improving fast paxos: being optimistic with no overhead. In *Pacific Rim Dependable Computing, Proceedings*, 2006.

[CBS07] B. Charron-Bost and A. Schiper. Harmful dogmas in fault tolerant distributed computing. *SIGACT News*, 38(1):53–61, 2007.

[CBS09] B. Charron-Bost and A. Schiper. The heard-of model: computing in distributed systems with benign faults. *Distributed Computing*, 22(1):49–71, 2009.

[CDG$^+$05] G. Chockler, M. Demirbas, S. Gilbert, C. Newport, and T. Nolte. Consensus and collision detectors in wireless ad hoc networks. In *Proceedings of the 24th Annual ACM Symposium on Principles of Distributed Computing (PODC'05)*, pages 197–206, New York, NY, USA, 2005. ACM.

[CF99] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[CHT96] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, June 1996.

[CKL+09] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche. Upright cluster services. In *SOSP '09: Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 277–290, New York, NY, USA, 2009. ACM.

[CKPS01] C. Cachin, K. Kursawe, F. Petzold, and V. Shoup. Secure and efficient asynchronous broadcast protocols. In *CRYPTO*, pages 524–541, 2001.

[CKS05] C. Cachin, K. Kursawe, and V. Shoup. Random oracles in constantinople: Practical asynchronous byzantine agreement using cryptography. *J. Cryptology*, 18(3):219–246, 2005.

[CL02] M. Castro and B. Liskov. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20(4):398–461, 2002.

[CML+06] J. Cowling, D. Myers, B. Liskov, R. Rodrigues, and L. Shrira. Hq replication: A hybrid quorum protocol for byzantine fault tolerance. In *7th USENIX Symposium on Operating Systems Design and Implementation (OSDI'06)*, Seattle, Washington, November 2006.

[CP02] C. Cachin and J. A. Poritz. Secure intrusion-tolerant replication on the internet. In *Dependable Systems and Networks (DSN'02)*, pages 167–176, 2002.

[CSS03] D. Cavin, Y. Sasson, and A. Schiper. FRANC: A Lightweight Java Framework for Wireless Multihop Communication. Technical report, EPFL, 2003.

[CSS04] D. Cavin, Y. Sasson, and A. Schiper. Consensus with Unknown Participants or Fundamental Self-Organization. In *ADHOC-NOW 2004*, pages 135–148, 2004.

[CSS05] D. Cavin, Y. Sasson, and A. Schiper. Reaching Agreement with Unknown Participants in Mobile Self-Organized Networks in Spite of Process Crashes. Technical Report IC/2005/026, EPFL, 2005.

[CT96] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CWA+09] A. Clement, E. Wong, L. Alvisi, M. Dahlin, and M. Marchetti. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *6th USENIX Symposium on Networked Systems Design and Implementation (NSDI'09)*, pages 153–168, Berkeley, CA, USA, 2009. USENIX Association.

[DBFC07] W. S. Dantas, A. N. Bessani, J. d. S. Fraga, and M. Correia. Evaluating byzantine quorum systems. In *Proceedings of the 26th IEEE International Symposium on Reliable Distributed Systems (SRDS'07)*, pages 253–264, Washington, DC, USA, 2007. IEEE Computer Society.

[DDS87] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[DG02] P. Dutta and R. Guerraoui. Fast indulgent consensus with zero degradation. In *4th European Dependable Computing Conference (EDCC'02)*, pages 191–208, 2002.

[DGDF$^+$08] C. Delporte-Gallet, S. Devismes, H. Fauconnier, F. Petit, and S. Toueg. With finite memory consensus is easier than reliable broadcast. In *12th International Conference on Principles of Distributed Systems (OPODIS'08)*, pages 41–57, 2008.

[DGK07] P. Dutta, R. Guerraoui, and I. Keidar. The overhead of consensus failure recovery. *Distributed Computing*, 19(5-6):373–386, April 2007.

[DGL05] P. Dutta, R. Guerraoui, and L. Lamport. How fast can eventual synchrony lead to consensus? In *Dependable Systems and Networks (DSN'05)*, pages 22–27, Los Alamitos, CA, USA, 2005.

[DLS88] C. Dwork, N. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[DRS90] D. Dolev, R. Reischuk, and H. R. Strong. Early stopping in byzantine agreement. *J. ACM*, 37(4):720–741, 1990.

[DS83] D. Dolev and H. R. Strong. Authenticated algorithms for byzantine agreement. *SIAM J. Comput.*, 12(4):656–666, 1983.

[DS97] A. Doudou and A. Schiper. Muteness failure detectors for consensus with Byzantine processes. TR 97/230, EPFL, Dept d'Informatique, October 1997.

[DS98] A. Doudou and A. Schiper. Muteness detectors for consensus with Byzantine processes (Brief Announcement). In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, Puerto Vallarta, Mexico, July 1998.

[DS06] D. Dobre and N. Suri. One-step consensus with zero-degradation. In *Dependable Systems and Networks (DSN'06)*, pages 137–146, 2006.

[DSU04] X. Défago, A. Schiper, and P. Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[Fis83] M. J. Fischer. The consensus problem in unreliable distributed systems (a brief survey). In *Fundamentals of Computation Theory*, pages 127–140, 1983.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[Gaf98] E. Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the 17th Annual ACM Symposium on Principles of Distributed Computing (PODC'98)*, pages 143–152, Puerto Vallarta, Mexico, 1998. ACM Press.

[GKQV10] R. Guerraoui, N. Knezevic, V. Quéma, and M. Vukolic. The next 700 bft protocols. In *EuroSys*, pages 363–376, 2010.

[Gro97] I. W. Group. *Wireless LAN Medium Access Control (MAC) and Physical Layer (PHY) Specifications.* IEEE Strandard 802.11-1997, New York, NY, 1997.

[GT07] F. Greve and S. Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *Dependable Systems and Networks (DSN'07)*, pages 82–91, Washington, DC, USA, 2007. IEEE Computer Society.

[Gue00] R. Guerraoui. Indulgent algorithms (preliminary version). In *Proceedings of the 19th Annual ACM Symposium on Principles of Distributed Computing (PODC'00)*, pages 289–297, 2000.

[GV07] R. Guerraoui and M. VukoliĆ. Refined quorum systems. In *Proceedings of the 26th Annual ACM Symposium on Principles of Distributed Computing (PODC'07)*, pages 119–128, New York, NY, USA, 2007. ACM.

[Hen99] J. Hennessy. The future of systems research. *Computer*, 32(8):27–33, 1999.

[HR99] M. Hurfin and M. Raynal. A simple and fast asynchronous consensus protocol based on a weak failure detector. *Distrib. Comput.*, 12(4):209–223, 1999.

[HS07] M. Hutle and A. Schiper. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In *Dependable Systems and Networks (DSN'07)*, pages 92–10. IEEE, June 2007.

[HT93] V. Hadzilacos and S. Toueg. Fault-tolerant broadcasts and related problems. In S. Mullender, editor, *Distributed systems (2nd Ed.)*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[HW05] M. Hutle and J. Widder. On the possibility and the impossibility of message-driven self-stabilizing failure detection. In *Proceedings of the 7th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'05)*, pages 153–170, 2005. Appeared also as Brief Announcement at PODC'05.

[KAD+07] R. Kotla, L. Alvisi, M. Dahlin, A. Clement, and E. Wong. Zyzzyva: speculative byzantine fault tolerance. *SIGOPS Oper. Syst. Rev.*, 41(6):45–58, 2007.

[KMMS97] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Solving consensus in a byzantine environment using an unreliable fault detector. In *1st International Conference on Principles of Distributed Systems (OPODIS'97)*, pages 61–75, Chantilly, France, December 1997.

[KS06] I. Keidar and A. Shraer. Timeliness, failure-detectors, and consensus performance. In *Proceedings of the 25th Annual ACM Symposium on Principles of Distributed Computing (PODC'06)*, pages 169–178, New York, NY, USA, 2006. ACM Press.

[KS07] A. Khelil and N. Suri. Gossiping: Adaptive and reliable broadcasting in manets. In *Third Latin-American Sympo-*

sium on Dependable Computing (LADC'07), pages 123–141, 2007.

[Lam78] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, 1978.

[Lam84] L. Lamport. Using time instead of timeout for fault-tolerant distributed systems. *ACM Trans. Program. Lang. Syst.*, 6(2):254–280, 1984.

[Lam98] L. Lamport. The part-time parliament. *ACM Transactions on Computer Systems*, 16(2):133–169, May 1998.

[Lam01] L. Lamport. Paxos made simple. *ACM SIGACT News*, 32(4):18–25, 2001.

[Lam05] L. Lamport. Fast paxos. Technical Report MSR-TR-2005-12, Microsoft Research, 2005.

[Lam06] L. Lamport. Fast paxos. *Distributed Computing*, 19(2):79–103, 2006.

[Lam09] L. Lamport. State-Machine Reconfiguration: Past, Present, and the Cloudy Future. DISC Workshop on Theoretical Aspects of Dynamic Distributed Systems, September 2009.

[LSP82] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[Lyn96] N. Lynch. *Distributed Algorithms.* Morgan Kaufman, 1996.

[MA06] J.-P. Martin and L. Alvisi. Fast byzantine consensus. *Transactions on Dependable and Secure Computing*, 3(3):202–214, 2006.

[MCS$^+$06] M. Mohsin, D. Cavin, Y. Sasson, R. Prakash, and A. Schiper. Reliable broadcast in wireless mobile ad hoc networks. In *39th Annual Hawaii International Conference on System Sciences (HICSS'06)*, page 233.1, Washington, DC, USA, 2006. IEEE Computer Society.

[MHS09] Z. Milosevic, M. Hutle, and A. Schiper. Unifying Byzantine consensus algorithms with weak interactive consistency. In *13th International Conference on Principles of Distributed Systems (OPODIS'09)*, 2009.

[MNCV06] H. Moniz, N. Neves, M. Correia, and P. Verissimo. Randomized intrusion-tolerant asynchronous services. In *Dependable Systems and Networks (DSN'06)*, pages 568–577, 2006.

[MR98] D. Malkhi and M. Reiter. Byzantine quorum system. *Distributed Computing*, 11:569–578, 1998.

[MR01] A. Mostéfaoui and M. Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, 2001.

[MW88] Y. Moses and O. Waarts. Coordinated traversal: (t + 1)-round byzantine agreement in polynomial time. In *29th Annual Symposium on Foundations of Computer Science (FOCS'88)*, pages 246–255, 1988.

[OL88] B. M. Oki and B. H. Liskov. Viewstamped replication: A new primary copy method to support highly-available distributed systems. In *Proceedings of the 7th Annual ACM Symposium on Principles of Distributed Computing (PODC'88)*, pages 8–17, New York, NY, USA, 1988. ACM.

[PLL97] R. D. Prisco, B. Lampson, and N. Lynch. Revisiting the paxos algorithm. In *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, volume LNCS 1320/1997, pages 111–125. Springer-Verlag, 1997.

[PS02] F. Pedone and A. Schiper. Handling message semantics with generic broadcast protocols. *Distrib. Comput.*, 15(2):97–107, 2002.

[PSL80] M. Pease, R. Shostak, and L. Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[Rab83] M. Rabin. Randomized Byzantine generals. In *Symposium on Foundations of Computer Science*, pages 403–409, 1983.

[SBD$^+$10] M. Serafini, P. Bokor, D. Dobre, M. Majuntke, and N. Suri. Scrooge: Reducing the costs of fast byzantine replication in presence of unresponsive replicas. In *Dependable Systems and Networks (DSN'10)*, pages 353–362, 2010.

[Sch90] F. B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[Sch93] F. B. Schneider. Replication Management using the State-Machine Approach. In S. Mullender, editor, *Distributed Systems*, pages 169–197. ACM Press, 1993.

[Sch97] A. Schiper. Early consensus in an asynchronous system with a weak failure detector. *Distributed Computing*, 10(3):149–157, April 1997.

[Sch09] A. Schiper. Distributed algorithms, 2009. Lecture Notes, EPFL, Switzerland.

[SCS02] Y. Sasson, D. Cavin, and A. Schiper. Probabilistic Broadcast for Flooding in Wireless Mobile Ad hoc Networks. Technical report, EPFL, 2002.

[SS08] M. Serafini and N. Suri. Reducing the costs of large-scale bft replication. In *2nd Workshop on Large-Scale Distributed Systems and Middleware (LADIS'08)*, pages 1–5, New York, NY, USA, 2008. ACM.

[ST87a] T. K. Srikanth and S. Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, 1987.

[ST87b] T. K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

[SvR08] Y. J. Song and R. van Renesse. Bosco: One-step Byzantine asynchronous consensus. In *Proceedings of the 22nd International Conference on Distributed Computing (DISC'08)*, pages 438–450, 2008.

[SW89] N. Santoro and P. Widmayer. Time is not a healer. In *Proceedings 6th Annual Symposium on Theor. Aspects of Computer Science (STACS'89)*, volume 349 of *LNCS*, pages 304–313, Paderborn, Germany, February 1989. Springer-Verlag.

[SW90] N. Santoro and P. Widmayer. Distributed function evaluation in the presence of transmission faults. In *SIGAL International Symposium on Algorithms*, pages 358–367, 1990.

[SW05] N. Santoro and P. Widmayer. Majority and unanimity in synchronous networks with ubiquitous dynamic faults. In *12th International Colloquium on Structural Information and Communication Complexity (SIROCCO'05)*, pages 262–276, 2005.

[SW07] N. Santoro and P. Widmayer. Agreement in synchronous networks with ubiquitous faults. *Theor. Comput. Sci.*, 384(2-3):232–249, 2007.

[Tou84] S. Toueg. Randomized byzantine agreements. In *Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing (PODC'84)*, pages 163–178, New York, NY, USA, 1984. ACM.

[VCBL09] G. S. Veronese, M. Correia, A. N. Bessani, and L. C. Lung. Spin one's wheels? byzantine fault tolerance with a spinning primary. In *Proceedings of the 28th IEEE International Symposium on Reliable Distributed Systems (SRDS'09)*, pages 135–144, Washington, DC, USA, 2009. IEEE Computer Society.

[VE05] E. W. Vollset and P. D. Ezhilchelvan. Design and performance-study of crash-tolerant protocols for broadcasting and reaching consensus in manets. In *Proceedings of the 24th IEEE International Symposium on Reliable Distributed Systems (SRDS'05)*, pages 166–178, Washington, DC, USA, 2005. IEEE Computer Society.

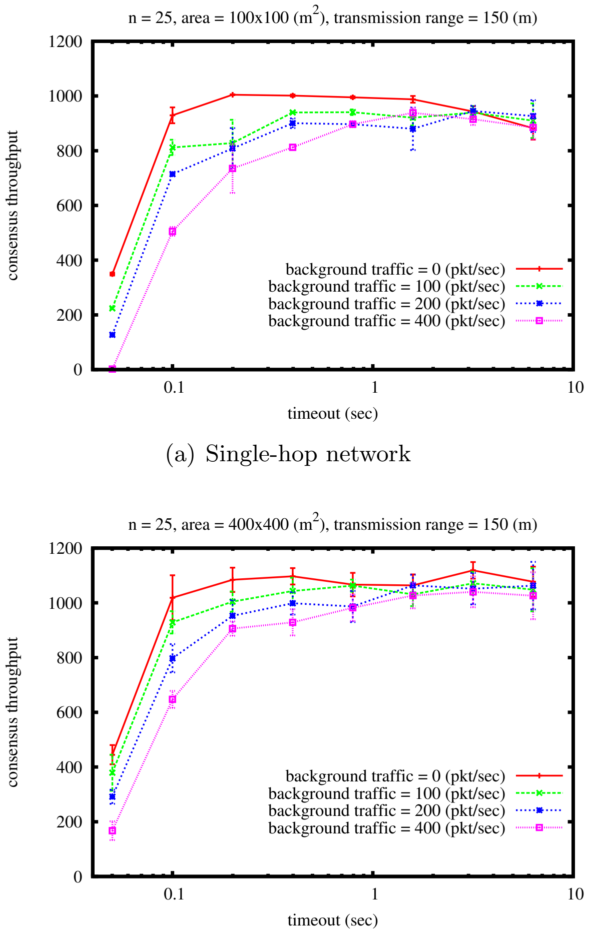
[WCYR07] W. Wu, J. Cao, J. Yang, and M. Raynal. Design and Performance Evaluation of Efficient Consensus Protocols for Mobile Ad Hoc Networks. *IEEE Transactions on Computers*, 56(8):1055–1070, August 2007.

[WSVS08] T. Wood, R. Singh, A. Venkataramani, and P. Shenoy. ZZ: Cheap practical BFT using virtualization. Technical Report TR14-08, University of Massachusetts, 2008.

[YMV$^+$03] J. Yin, J.-P. Martin, A. Venkataramani, L. Alvisi, and M. Dahlin. Separating agreement from execution for byzantine fault tolerant services. *SIGOPS Oper. Syst. Rev.*, 37(5):253–267, 2003.

Appendix A

A.1 Impact of background traffic

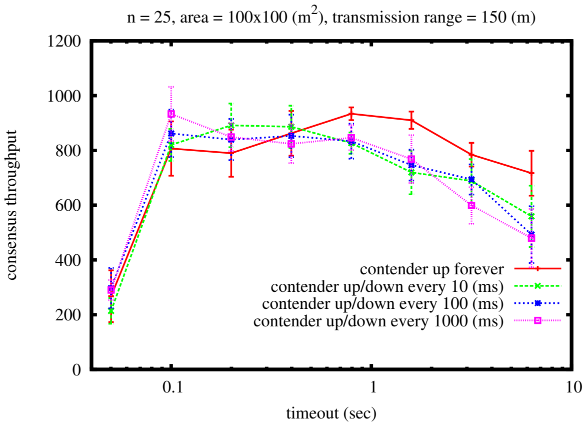


(a) Single-hop network

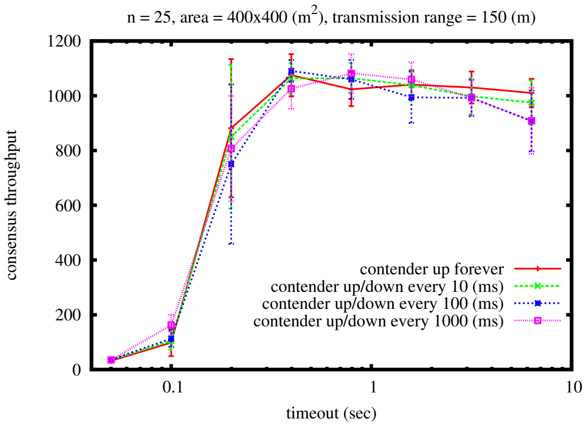
(b) Multi-hop network

Figure A.1: Impact of background traffic on consensus throughput.

A.2 Impact of coordinator crash



(a) Single-hop network



(b) Multi-hop network

Figure A.2: Impact of coordinator crash on consensus throughput.

Curriculum Vitae

I was born in Tabriz, Iran in 1978. I obtained a diploma in Mathematics and Physics in Tehran in 1996. I studied Computer Engineering and then Mathematics and Statistics at the *National University of Tehran*.

I moved to Switzerland in September 2001. I started studying Computer Science at EPFL in October 2001. I then graduated from EPFL in April 2006 with a B.Sc. and a M.Sc. degrees in Computer Science. I accomplished my master's thesis, entitled "Efficient Semi-structured Queries in Scala using XQuery Shipping", at the Programming Methods Laboratory (LAMP, EPFL) under the supervision of Professor Martin Odersky. I carried out two summer internships during my study at EPFL.

Since April 2006, I have been working at the Distributed Systems Laboratory (LSR, EPFL) as a research and teaching assistant and as a PhD student under the supervision of Professor André Schiper.