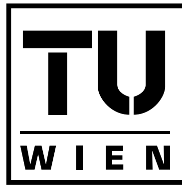
Die approbierte Originalversion dieser Dissertation ist an der Hauptbibliothek der Technischen Universität Wien aufgestellt (http://www.ub.tuwien.ac.at).

The approved original version of this thesis is available at the main library of the Vienna University of Technology (http://www.ub.tuwien.ac.at/englweb/).



DISSERTATION

Distributed Computing in the Presence of Bounded Asynchrony

ausgeführt zum Zwecke der Erlangung des akademischen Grades
eines Doktors der technischen Wissenschaften
unter der Leitung von

Univ.Prof. Dr. Ulrich Schmid
Inst.-Nr. E182/2
Institut für Technische Informatik
Embedded Computing Systems

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.-Ing. Josef Widder
Matr.-Nr. 9625114
Meidlgasse 41/4/4
A-1110 Wien
Europäische Union

Wien, im Mai 2004 ______________________

Verteilte Algorithmen unter eingeschränkter Asynchronität

Diese Dissertation untersucht verschiedene Aspekte des Θ-Modells, einem zeitfreien Systemmodell für verteilte Systeme. Die zugrunde liegende Annahme ist die Existenz eines begrenzten Verhältnisses der Übertragungsdauern von Nachrichten die gleichzeitig unterwegs sind. Das Modell wurde von Le Lann und Schmid eingeführt, die zeigten, dass das fundamentale *consensus* Problem in ihm eine Lösung besitzt.

Der erste Teil dieser Dissertation beschäftigt sich mit einer Neudefinition des Modells basierend auf zwei Modellen. Als erstes das *system model*, das ein sehr flexibles Zeitverhalten modelliert und daher eine hohe Abdeckung möglicher Zustände von realen Netzwerken hat. Das zweite ist das *computational model*, das der ursprünglichen Definition entspricht. Weiters wird gezeigt, dass die beiden Modelle gleich stark sind, dh. Probleme die eine Lösung in einem Modell haben, haben auch eine im anderen.

Der zweite große Abschnitt beschäftigt sich mit Algorithmen und deren Verhalten im Θ-Modell. Der grundlegende Algorithmus dient zur Uhrensynchronisation; auf ihm basierend werden dann andere Algorithmen aufgebaut. In diesem Teil wird auch mittels *strongly dependent decision problem* gezeigt, dass das Θ-Modell bezüglich Lösbarkeit von Problemen zu den synchronen gezählt werden muss, obwohl es keine obere Grenze von Übertragungsdauern voraussetzt.

Der dritte und letzte Abschnitt beschäftigt sich mit dem Problemkreis Netzwerkinitialisierung (*booting*). In der Theorie wird meist über dieses Problem hinweg gesehen, wahrscheinlich weil einfache Annahmen über das Zeitverhalten getroffen werden können, die dieses Problem einfach "weg definieren". Da in dieser Dissertation zeitfreie Modelle und Algorithmen untersucht werden, sind solche Annahmen nicht zulässig, weil darauf basierende Lösungen nicht mehr als zeitfrei bezeichnet werden können. Wie auch im zweiten Abschnitt beginnen wir mit Uhrensynchronisation in der Initialisierungsphase und stellen dann Lösungen von anderen Problemen vor, die darauf basieren.

Distributed Computing in the Presence of Bounded Asynchrony

This thesis investigates various aspects of the Θ-Model. The Θ-Model is a time free model of distributed systems which assumes that end-to-end delays of the fastest and slowest messages over the network are correlated. This relation is expressed by giving an upper bound Θ on the ratio of longest and shortest transmission times of messages which are simultaneously in transit. The model was introduced by Le Lann and Schmid, who showed that the Θ-Model is sufficiently strong to solve the fundamental yet not trivial problem of consensus. Their innovative results left room for improvement in the definition of the Θ-Model and raised some questions, including the amount of synchrony in the model and related to it what kind of problems have solution in it.

The first part of this thesis is dedicated to a refinement of the original definition of the Θ-Model. This is achieved by distinguishing two different models: The first one is the system model, which is very flexible regarding the timing of the described system. It should reach very high assumption coverage—the probability that the assumptions made in a model hold in real systems. The second model is the computational model, which is similar to the original definition of the Θ-Model. After that, we show that these two models are equivalent with respect to expressiveness. It hence follows that existing results, which rely upon the original definition of the Θ-Model, are valid in the novel system model as well.

The next major part introduces several algorithms and analyzes their behavior when executed in the Θ-Model. The basic algorithm is a clock synchronization algorithm whereupon several other algorithms—e.g. implementation of the perfect failure detector and atomic commitment—are devised. This part also gives an answer to the question of the amount of synchrony in the model. Using the strongly dependent decision problem—which is sort of a benchmark that separates synchronous from asynchronous models—it is shown that the Θ-Model must in fact be attributed as synchronous. This is quite surprising given that the system model does not stipulate an upper bound on message transmission delays.

The third part considers booting. The problem of system startup is often neglected in distributed computing theory. We believe that this is due to the fact that when real systems are considered, timed semantics are usually employed, which allow several simplifications of the booting problem. Since we consider a time free model, the problem of booting clock synchronization is particularly difficult because classic failure assumption (e.g. $f < n/3$) cannot be employed properly in the booting phase. Based upon a clock synchronization algorithm that properly handles booting, we show how to adapt solutions to other problems in distributed computing such that they provide (some of) their properties during booting as well.

Acknowledgments

I am grateful to my supervisor Ulrich Schmid who is the best teacher I ever had. He prepares an open and fruitful environment that allows young people (like I still pretend to be) to deliver scientific work at a level that can persist in the strong international competition. I thank Mehdi Jazayeri for his valuable comments and suggestions when grading this thesis. I am also grateful to Gérard Le Lann who invited me to work with him at INRIA Rocquencourt for two months. During this stay I learned very much, not limited to predictability of distributed systems. Further I thank Martin Hutle for proof reading drafts of this thesis. His remarks helped me improving the presentation of the results. Discussions with him give me deeper insight into many challenges we are facing.

Ich danke meiner Mutter Edith Widder und meinen Großeltern für das Schaffen eines familiären Umfeldes das mir mein Studium so leicht gemacht hat. Dank auch an meine Freunde Andi, Josef, Martin, Migo, und Tobi.

In Erinnerung an meinen Vater Josef Widder.

Supported by the Austrian BM:vit FIT-IT project DCBA, project no. 808198.

Human nature is such that simple, easy to understand solutions are spontaneously favoured (and selected). However, with distributed real-time applications, the question is whether such an instinctive attitude is appropriate...

Gérard Le Lann

Contents

1 Introduction **1**
1.1 Motivation for this Thesis . . . 2
1.2 Road Map . . . 3
1.3 Related Work . . . 4

2 Modeling Distributed Real-Time Systems **5**
2.1 Network of Queues . . . 5
2.2 Asynchronous Distributed Real-Time Systems . . . 7
2.3 Suitable Models for Real-Time Systems . . . 7

3 The Θ-Model for Distributed Real-Time Systems **11**
3.1 Physical System . . . 11
3.1.1 Faults . . . 11
3.1.2 Timing . . . 12
3.2 System Model . . . 12
3.3 Computational Model . . . 13
3.4 Equivalence of the Models . . . 14
3.5 Round Based Significant Uncertainty Ratio . . . 15
3.5.1 Byzantine Faults . . . 16
3.5.2 Crash Faults . . . 17
3.5.3 Clean-Crash Faults . . . 17
3.6 Related Work . . . 17

4 The Θ-Model Compared to other System Models **19**
4.1 Partial Synchrony . . . 19
4.2 Semi-Synchrony . . . 21
4.3 The Archimedean Assumption . . . 22

5 Implementation Issues **23**
5.1 Design Immersion . . . 23
5.2 Assumption Coverage . . . 23
5.3 Real-Time Scheduling . . . 24
5.4 Experimental Results . . . 24
5.5 Deterministic Ethernet . . . 25
5.6 Algorithms at Application Level . . . 25

5.7 Bounded Drift Clocks . . . 26

6 Selected Algorithms in the Θ-Model 27
6.1 Clock Synchronization . . . 27
6.1.1 Byzantine Case . . . 28
6.1.2 Clock Synchronization Properties . . . 30
6.1.3 Restricted Failure Modes . . . 33
6.2 Failure Detection . . . 37
6.3 Synchrony of the Θ-Model . . . 39
6.3.1 The Strongly Dependent Decision Problem . . . 40
6.3.2 Fault-Tolerant Broadcast . . . 42
6.3.3 Simulating Lock-Step . . . 44
6.4 Non-Blocking Atomic Commitment . . . 46
6.4.1 Clean-Crash Faults . . . 47
6.4.2 Crashes . . . 49
6.5 Related Work . . . 50
6.5.1 Clock Synchronization . . . 50
6.5.2 Unreliable Failure Detectors . . . 51
6.5.3 Synchronizers . . . 51
6.5.4 Non-Blocking Atomic Commit . . . 52

7 Booting Clock Synchronization 53
7.1 Perception Based Failure Model . . . 54
7.1.1 Execution Model . . . 54
7.1.2 Model of the Startup Phase . . . 58
7.1.3 Physical Failure Model . . . 59
7.1.4 Perception Failure Model . . . 60
7.2 The Algorithm . . . 65
7.3 Mapping to the Perception based Execution Model . . . 67
7.4 From Early to Degraded Mode . . . 68
7.5 From Degraded to Normal Mode . . . 74
7.6 Related Work . . . 81

8 Booting Θ-Algorithms 83
8.1 Eventually Perfect Failure Detector . . . 83
8.2 General Considerations . . . 85
8.3 Lock-Step . . . 85
8.4 Atomic Multicast . . . 87
8.5 Non-Blocking Atomic Commitment . . . 89

9 Conclusions 91
9.1 Required Synchrony for Consensus . . . 91
9.2 Clock Synchronization Implementation . . . 92
9.3 Dependable Real-Time Systems . . . 92
9.4 Outlook on Further Research Directions . . . 93

Chapter 1

Introduction

A distributed system consists of a collection of autonomous computers linked by a network such that these computers can coordinate their activities and share their resources. They employ distributed algorithms for this purpose which consist of multiple processes that are executed concurrently on the multiple processors. In order to exhaustively describe the system's behavior it is usually necessary to analyze these algorithms and prove their properties mathematically.

In sharp contrast to single computer systems, the fault of a single component does not necessarily lead to a failure of the whole system. Distributed algorithms should hence be able to tolerate a given number of faulty remote processes or links. (These numbers must usually be derived statistically from hardware and software error probabilities.) Adding the notion of fault tolerance to distributed systems renders many problems extremely difficult to handle, however. Classic results even show that fundamental problems like agreeing on a common value—consensus—is impossible in certain settings; cp. the well known FLP result [42]. These settings include the type of faults (e.g. computers that crash or send faulty values, lossy communications) and—of even more interest—the timing of the system. The mentioned FLP result, for example, just refers to one possible crash fault and to asynchronous systems, i.e. systems where message transmission and relative computer speeds are unbounded. Still: What kind of distributed systems really need such weak assumptions? Are there computers that are infinitely fast compared to others or are there networks where failure free sending of a message takes infinitely long? Since this is not the case, the assumption coverage of system models must be considered.

Models of distributed systems are formulated by simplifying the observed objects and by describing their behavior by sets of rules. These abstractions are used to reason formally about system properties. Assumption coverage is a measure of how well the model fits the real target system. It is the probability that the assumptions in the model hold in the system. Returning to the FLP model, its coverage regarding timing assumptions is 1 in every real system such that solutions to problems in asynchronous systems can be transferred into all types of systems without decreasing coverage. Since there is no solution for agreement in asynchronous systems, however, additional assumptions (feasibility conditions) must be added in order to solve the problem. These

assumptions should be as weak as possible such that coverage can be maximized. Much work [29, 19] was devoted to this problem. Two major approaches can be distinguished here: (1) Adding synchrony and (2) adding information about failures. As a matter of fact these approaches are related to each other since implementing (2) is possible only by (1); nevertheless (2) is addressed in literature very often since it is a smart way to encapsulate synchrony in the asynchronous world.

Synchrony can be added by assumptions which are made about the timing behavior of a system or system components, e.g. bounds on message delays and computing speeds. Regardless of how values for the timing are derived, the probability that they hold during real operation must always be considered strictly less than 1. If the goal is maximizing assumption coverage, the means for achieving this is weakening the assumptions as far as possible while maintaining the solvability of the required problems (under the given set of assumptions).

But lets now turn to a special property of distributed computing problems: Real-time requirements. Real-time problems arise in applications where not only correct responses are required, but these responses also have to meet deadlines. In the real-time community exists a widespread believe nowadays that synchronous (timed) system models are required in order to provide solutions to real-time problems. It should be clear by now that these solutions have the drawback that their assumption coverage is necessarily smaller than those of time free models. Therefore approaches are required that reconcile real-time and maximal coverage. Such an approach is the design immersion principle [65, 66] which just states: Take a time free solution and analyze it (using schedulability analysis) on a given system that employs optimal online scheduling algorithms. If this analysis reveals that the solution executed on the given system meets all its deadlines we have a real-time system. This approach is preferable since, in order to maximize safety (nothing bad ever happens) and coverage, least assumptions should be made and synchrony should be introduced as late as possible during the development life cycle. More abstractly, when immersing a time free solution into a real distributed system, we say that the system's timing properties emerge. Thus it is possible to design time free real-time systems. This thesis is devoted to this branch of research.

1.1 Motivation for this Thesis

As one possible timing model for time free real-time applications, Le Lann and Schmid [67, 68] introduced the Θ-Model. In this model, Θ is a measure of asynchrony in distributed systems; namely an upper bound on the ratio of the longest and shortest end-to-end message transmission times. To demonstrate its suitability for algorithms solving asynchronous real-time distributed computing problems, a major task is to devise solutions to the most fundamental agreement problems like FD based consensus, atomic broadcast and atomic commitment.

Since the problem of handling system booting becomes difficult in asynchronous—i.e. time free—designs, as already discussed in [106], the behavior of the mentioned algorithms will be explored during booting as well.

In addition, a refinement of the original Θ-Model is presented, which is particularly promising in terms of assumption coverage. Explicit upper bounds are dismissed by just referring to longest message transmission at some given point in time which effectively leads to time dependent bounds (i.e. that may vary over time). It is then shown that the new definition of the model has the same expressive power than the original definition such that existing results remain valid. Moreover, the analysis of algorithms may still be conducted in the simple to handle original model and "mechanically" translated into the weaker model.

We compare the Θ-Model to partially synchronous and semi-synchronous models, by opposing the respective assumptions and discussing their relation. We also compare their respective power by examining solutions to some specific distributed computing problems, like the SDD problem.

1.2 Road Map

The remainder of this thesis is divided into three major parts: The Θ-Model, algorithms, and system booting.

It starts with Section 2 where properties of real systems are discussed and several models are examined w.r.t. their suitability for describing those. Based upon these results, we introduce the Θ-Model in Section 3, where we provide two models: The system model in Section 3.2 models the timing behavior of a distributed system with high coverage. The second model is an easy to handle computational model which is introduced in Section 3.3. We will show that both models are equivalent regarding expressive power such that results of the computational model can easily be transferred to the high coverage system model of Section 3.2. Since, by then, we have introduced the model we are ready to compare it to other well known models in Section 4. In Section 5 we discuss several practical issues related to the Θ-Model.

The second major part can be found in Section 6 and is devoted to algorithms for the Θ-Model. We introduce fundamental building blocks first which are required by any of our high-level algorithms, namely clock synchronization in Section 6.1 and perfect failure detection (for solving consensus) in Section 6.2. Section 6.3 is devoted to the more theory related question of the degree of synchrony in the Θ-Model. To accomplish this, a solution of the strongly dependent decision problem is given which shows that the Θ-Model is equivalent to synchronous models in this respect. Based upon those foundations we give an algorithm for atomic broadcast in Section 6.3.2 and a lock-step simulation in Section 6.3.3. This simulation allows to solve agreement problems even in presence of the most severe type of faults, i.e. Byzantine faults. Since non-blocking atomic commitment is not reducible to consensus or atomic broadcast in asynchronous systems, we finally give solutions to this problem in Section 6.4 as well.

It is usually (implicitly) assumed in distributed computing theory that all processes are initially up such that they do not miss messages sent by other correct processes. In timed systems it is convenient to use information on time for booting the system – at the cost of reduced coverage though. Since we consider time free algorithms we must handle system booting in a time free manner. The third major part is hence devoted to

this problem: A solution to the pivotal clock synchronization algorithm that works also during booting is presented in Section 7. This section also reviews the perception based failure model, which underlies the analysis of our clock synchronization algorithm. Booting in the context of other distributed algorithms is discussed in Section 8.

1.3 Related Work

The books by Coulouris et al. [23], Tanenbaum et al. [97] and the book edited by Mullender [77] are good introductions into basic problems and concepts of distributed systems. More related to the work that is presented here are the books by Lynch [71], Attiya and Welch [9] and Tel [98] as they concentrate on the theoretical aspects of distributed systems and algorithms. Basic principles of real-time systems are presented in the books by Kopetz [59] and Veríssimo et al. [101].

This thesis focuses on the Θ-Model and its ability to solve real-time problems. The results presented here build upon the work by Le Lann and Schmid, who introduced the Θ-Model as well as an implementation of the perfect failure detector [67] and discussed its ability to maximize coverage [68]. Part of our results were/will be published in a number of papers: Clock synchronization algorithms that support booting are presented in [106, 109]. Based on these algorithms, a refinement of the failure detector implementation is given in [108], which can be used during booting as well. For a variant of the Θ-Model, were the synchrony assumptions hold just after some possibly unknown global stabilization time, a simulation for Byzantine consensus is presented in [107]. In [55] two self-stabilizing failure detector implementations are devised. The relation of the Θ-Model to real-time networks is examined in [52]: The timing behavior of Θ-algorithms is observed when executed in an architecture built upon deterministic Ethernet.

Chapter 2

Modeling Distributed Real-Time Systems

Since any solution to a problem in distributed real-time computing is just as good as its underlying model, a crucial part in the design of dependable real-time systems is the choice of the right model. When choosing a model, several topics must be addressed. First of all, one has to examine what to describe, i.e. the type of systems the devised algorithms shall eventually be executed on. A suitable model should adequately describe the core properties of those systems. Among those are queuing phenomenons as discussed in Section 2.1, which severely affect the system timing.

The problems that have to be solved by the algorithms are obviously another important topic that must be addressed when choosing a model. We discuss novel application domains for real-time systems in Section 2.2.

After this we shortly review existing approaches which aim at these novel application domains in Section 2.3 and discuss whether these are suitable for the problems we want to address.

2.1 Network of Queues

A message transmission over a network consists of local message preparation at the sender, transmission over the link, and local receive computation at the receiver. This is of course an abstraction which hides away that messages go through various queues at the sender and receiver such that the sojourn times in queues—depending on scheduling characteristics—must be considered as important part of the message transmission delay δ.

In Figure 2.1 it is depicted how a distributed system can be modeled as a network of queues. All messages that drop in over one of the $n-1$ incoming links of a processor must eventually be processed by the single CPU. Every message that arrives while the CPU processes former ones must hence be put into the CPU queue for later processing. In addition, all messages produced by the CPU must be scheduled for transmission over every outgoing link. Messages that find an outgoing link busy must hence be put into the send queue of the link's communication controller for later transmission.

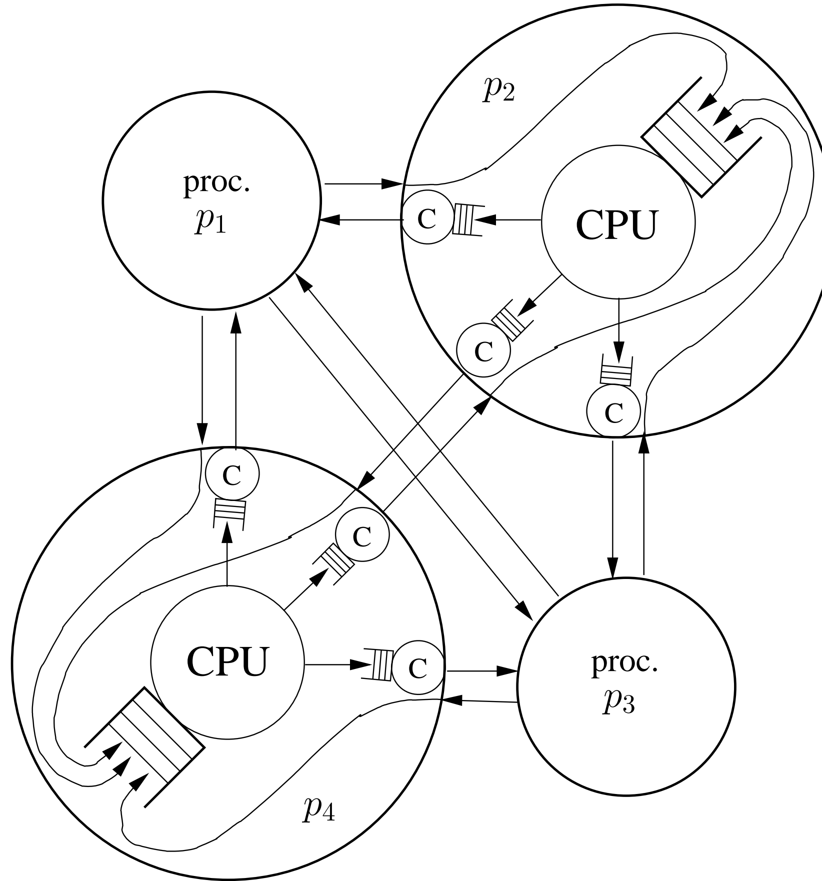


Figure 2.1. A simple queuing system representation of a fully connected distributed system of 4 processors

As Le Lann and Schmid [67] observe, the transmission delay (between two correct processes p and q) $\delta_{pq} = d_{pq} + \omega_{pq}$ consists of two parts: A fixed part $d_{pq} > 0$ which is determined by process speeds and data transmission characteristics and a variable part $\omega_{pq} \geq 0$ which captures all variations of δ_{pq}; including:

- precedences, resource sharing and contention, with or without resource preemption, which creates waiting queues,
- varying (application-induced) load,
- varying process execution times (which may depend on actual values of input parameters),
- occurrence of failures.

Especially in networks with shared channel topologies (like Ethernet), these points do not vary independently at different nodes. The shared channel makes it impossible for the adversary to slow down just some messages, and hence to violate some upper bound τ^+ without slowing down all messages simultaneously. It follows that the lower bound on messages delays τ^- cannot be attained simultaneously during such periods. Hence, the Θ-Model (cp. Section 3) does not bound τ^+ and τ^- but just the ratio of these bounds $\Theta = \tau^+/\tau^-$. Regarding coverage this is advantageous for systems where the smallest delays (τ^-) are increased at least by c/Θ whenever the longest delays (τ^+) increase by an amount of c: Obviously, Θ cannot be violated here.

2.2 Asynchronous Distributed Real-Time Systems

Recently, new application domains, like space, autonomy and artificial intelligence etc. became targets for distributed real-time systems [56]. A peculiarity of these new applications is the timing behavior induced by the uncertainty regarding future operational environments. Requirements of such applications are hence different from classical application domains of real-time systems, like engine control or industrial automation at machine level. These applications normally comprise very simple control tasks executed on dedicated processors with constant computation load and therefore predictable timing behavior, such that synchronous timing models are suitable, i.e. have sufficiently large coverage (the probability that the assumptions in the model hold during real operation).

For the above mentioned new application domains new system models must be devised since synchronous models of computation do not apply. One reason for this are uncertain environments: Consider long lived space borne applications, for example, where timing uncertainty is induced just by the unpredictable environment. Another source for asynchrony are intensive on demand computations. Choosing timing bounds for normal operation according to rare computation requirements is costly and often not even required from a safety point of view. Finally using COTS (commercially of the shelf) components significantly reduces costs which opens new application domains for real-time systems. Unfortunately—due to pipelines, caches etc.—modern COTS processors do have quite unpredictable timing behavior. Therefore weaker models of computation should be devised [51, 66] in order to increase the coverage.

2.3 Suitable Models for Real-Time Systems

We now shortly review what kinds of weak timing models have been devised by the distributed computing community, and how suitable they are for real-time distributed applications. It is very well known that, due to the fundamental FLP impossibility result [42], important agreement problems have no solution in the weakest timing model, i.e. pure asynchrony. Since our goal are models that are as weak as possible, we take FLP as starting point and review what can be added to the asynchronous model in order to solve agreement problems which are required as building blocks for most real-time control applications.

Lets start with the most abstract approach taken by Chandra and Toueg [19], who introduced the concept of failure detectors (FDs) and showed that even unreliable FDs are sufficient to solve consensus in asynchronous systems. We regard their contribution as twofold. On one hand they formulated what kind of information on failures is required by processes in asynchronous environments in order to solve consensus. Second, FDs are time free constructs which hide synchrony assumptions. As shown by Hermant and Le Lann [51], this second contribution can be employed very efficiently in real-time systems by recognizing that implementing FDs can be done in lower network layers. As already exploited in many clock synchronization based application, lower level uncertainty is much smaller than application level uncertainty, i.e. clocks can be synchronized closer than it would be possible on application level (where they are used). FD implementations can, similarly, be devised which detect failures fast, i.e. in shorter times than application level messages may travel in the worst case.

Nevertheless, FDs are abstract, time free constructs. For distributed real-time systems we must guarantee that all deadlines are met, i.e. we have to model the synchrony of our system more explicitly. Note that we may still refer to FD based solutions, but we have to give time bounds on failure detection times as well as on termination of agreement algorithms. We therefore now discuss some timing models which are neither totally asynchronous nor totally synchronous.

Dolev, Dwork and Stockmeyer [29] investigated how much synchronism has to be added to asynchronous systems in order to solve consensus. They identified five synchrony parameters (processors, communication, message order, broadcast facilities and atomicity of actions) which can be varied into 32 different *partially synchronous* system models. For each of those, they investigated whether consensus is solvable or not. Solutions for consensus in such system models are given in the paper by Dwork, Lynch and Stockmeyer [37], where two sources for uncertainty—computational speed and transmission times—are considered. They postulated Δ as (unknown) upper bound on message delay (on the network) and Φ as upper bound on relative computational speeds. Although this model is weaker than the synchronous model, it has many hidden assumptions which do not seem suitable for real-time applications. For example delivering of a set of message takes one step regardless how large this set is. This totally ignores timing uncertainty induced by different queuing schemes and varying network load during operation. One step is required as well to send a message point-to-point. We believe—and argue this in more detail in Section 4.1 and Section 4.2—that such modeling is not suitable for real-time requirements since the hidden assumption are too restrictive for the targeted application domains.

As mentioned above, new application domains for real-time systems [56] triggered some more recent research on novel timing models. We will discuss two well known approaches here, namely the *timed asynchronous distributed system model* TA [24, 74] and the *timely computing base model* TCB [99, 100, 17].

The TA model assumes that during system operation there exist sufficiently long periods during which no faults occur and liveness can be achieved. At first sight this appears similar to the partial synchronous models [37] where it is assumed that, after some stabilization time, the system is synchronous. Partial synchronous models

describe systems where a known bound exists and eventually holds, however, whereas in TA some δ on the message delays is *defined*. How the value for δ is chosen is application dependent. One approach proposed in [24, 74] is that if the system should react within D real-time units, and the protocol requires a message chain consisting of k messages in order to react, then $\delta \triangleq D/k$ should be chosen. From this definition follows the definition of a timing fault, i.e. a message transmission that requires more than δ time. This is, in the best case, just a partial solution to a real-time problem. In fact δ is not an upper bound, but just some value for which it is assumed that "most" messages are faster than δ. The solution of a real-time problem would require a guarantee that δ holds sufficiently often, but no solution to this problem is given in [24, 74].

The TCB takes the same approach: Definition of timing properties. But here a whole subsystem, the timely computing base, is defined. Whenever the application requires something to happen timely, the TCB is used (for e.g. message transmission, local timeout). A TCB is defined to be independent of the asynchronous part of the system (interposition). The TCB is protected from faults that could violate timing (shielding). And its complexity is such that verifiable mechanisms can be implemented (validation). The problem, however, is that the description of the TCB [99] does not provide "feasibility conditions" like how often it may be used to transmit timely messages, while in the description of an example TCB implementation [17] it is argued that it cannot be overloaded since load can be controlled. Again, this is in the best case just a partial solution to a real-time problem since deriving rules on message arrivals would be required to ensure that the TCB is in fact timely. No such rules are presented in [99, 100, 17].

Both, TA and TCB are timed system models, i.e. local information on time is used to detect remote performance faults. The question arises whether this is required. It is often argued that such faults have to be detected in order to take actions that prevent hazards, i.e. to ensure safety. And it is believed that using local information on time provided by clocks is required to do so. The design immersion principle, introduced by Le Lann [51, 66] takes a different approach. The goals that have to be reached by a real-time system are safety, liveness, and timeliness. Distributed computing theory tells us that, for many problems, safety and liveness can be reached even in asynchronous systems, or at least in some sort of weak partial synchronous models. Design immersion describes a system engineering process where a design is made based on the weakest synchrony assumptions possible and formally proved to meet safety requirements (nothing bad ever happens) and liveness requirements (eventually something good happens). Only after this, the solution is immersed into a real target proved (e.g. via scheduling analysis) to satisfy certain timeliness requirements in this target system as well.

The advantage of this approach is increased coverage: It is more likely that safety and liveness hold in time free designs than in designs resting directly on the assumed timing properties of the target system: Improved coverage means that there are system states where the immersed solution still delivers some service while the direct solution does not. In order to investigate this we have to consider exceptional cases: Consider periods where the target system's behavior deviates from the expected one, which cannot be

ruled out as 100% coverage is impossible to reach, especially when missions with long operation times in not completely predictable environments have to be considered. In such periods a design resting on weaker assumptions may still deliver safety and liveness, while just timeliness is lost. A design directly resting on the target system may lose safety and liveness along with timeliness. Note that in such periods it is impossible for any solution to ensure timeliness, such that the best one can hope for is maintaining safety and liveness (properties which can be ensured even in purely asynchronous systems according to distributed computing theory).

The question, however, arises what kind of weak models are suitable for design immersion. There are two requirements for such systems: (1) important problems (like consensus) must have a solution in the model, (2) the model's timing assumptions must be implementable in real systems, and the assumptions should even hold in cases where the system does not behave as expected (in order to improve coverage further). We believe that the Θ-Model which was introduced by Le Lann and Schmid [67], is well suited for modeling real-time distributed systems. Unlike other partially synchronous models, the Θ-Model is sufficiently strong to solve consensus and related problems even in the absence of upper bounds on message transmission delays with links that need not provide FIFO (First-In First-Out) semantics. The apparent contradiction to FLP [42, 29] is resolved by replacing the upper bounds Δ and Φ by an upper bound Θ on the ratio of longest and shortest end-to-end transmission times. Computational speeds and transmission times are hence put together into end-to-end delays, which is the only duration which must adhere to our assumptions.

This kind of modeling has at least three important advantages: (1) for termination times of real-time applications only the end-to-end delays are relevant such that regarding anything else unnecessarily complicates analysis. (2) By modeling end-to-end delays we are less restrictive regarding relative computational speeds and transmission times: Only their sum must behave as expected such that a violation of either one does not necessarily lead to violation of our assumption. (3) Research results [68, 5] show that there exists a relation of upper and lower message transmission times in many systems: If a network is in a high load or even an overload state it is clear that upper message transmission times are larger than in low load periods. But in high load periods all messages must be queued at the network interfaces such that even the fastest transmissions become slower as well. In [52] we provide an example of how to build a system with a small value for Θ and how to prove timeliness properties of this system while executing time free algorithms.

Chapter 3

The Θ-Model for Distributed Real-Time Systems

After discussing the motivation for introducing new models we now formally define the objects of the Θ-Model and their interrelationships.

3.1 Physical System

We consider a system of n distributed processes[1] denoted as $p, q, \ldots$, which communicate through a reliable, error free and fully connected point-to-point network[2]. We assume that a non-faulty receiver of a message knows the sender. This assumption also includes that processes have distinct identifiers which can be ordered uniquely[3]. The communication channels between processes need not provide FIFO transmission, and there is no authentication service[4].

3.1.1 Faults

The most fundamental algorithms (clock synchronization) of this thesis deal with Byzantine nodes. In order do tolerate them we define the following threshold: Among the n processes there is a maximum of $f < n/3$ faulty ones. No assumption is made on the behavior of Byzantine faulty processes.

Problems like non-blocking atomic commitment (Section 6.4), reliable broadcast (Section 6.3.2), or strongly dependent decision (Section 6.3.1) are traditionally studied in the presence of crash respectively clean-crash faults only. By clean-crash fault we mean processes that do not crash in the middle of the execution of a statement. We require such behavior for statements like "send message to all" in order to ensure

[1] For conciseness we assume that every of the n processors in a system executes just one process.

[2] A model where several classes of link failures [109, 68] are considered is employed later, in Section 7.

[3] This assumption is required for atomic broadcast in Section 6.3.2 which requires that messages are ordered according to process identifiers.

[4] We do not consider authenticated algorithms since it is never guaranteed that malicious processes cannot break the authentication scheme. By devising solutions without authentication, our correctness proofs cannot be invalidated by this event.

consistent message transmissions[5]. Crash faulty processes on the other hand may crash at any time. In the presence of clean-crash faults we require systems where $f < n$. If crash faults are considered we require $f < n/2$. These numbers stem from the requirement to implement clock synchronization (compare Section 6.1.3).

In Section 7 we will use the perception based hybrid failure model [88] which incorporates various types of process and link failures. The model is discussed there in greater detail.

3.1.2 Timing

We consider time free algorithms, i.e. processes that do not have access to hardware clocks or an external time base. Regarding timing behavior we distinguish two models: The system model described in Section 3.2 which describes a dynamic timing behavior with no upper bounds upon message transmission delays δ. The ratio of longest and shortest message delays of messages that are simultaneously in transit, however, is bounded. Such timing behavior renders the analysis of such systems very complicated. To simplify the analysis we therefore introduce our computational model in Section 3.3 which just considers fixed but unknown upper and lower bounds on message transmission delays. We then show in Section 3.4 that these two models are equivalent, i.e. algorithms that have been shown to satisfy their safety and liveness requirements in the computational model also work in the system model.

Except the fundamental clock synchronization algorithm, all algorithms in this thesis will have a priori knowledge of some integer Ξ which is a function of the uncertainty ratio Θ (read on).

Regarding system booting we assume throughout Section 6 that all correct processes are initially up and listening to the network. We will see in Section 7 and Section 8, however, how to get rid of this assumption in a time free manner.

3.2 System Model

Processes communicate by message passing. The time interval a message m is in transit consists of three parts: Local message preparation (includes queuing) at the sender, transmission over the link, and local receive computation (includes queuing) at the receiver. We denote t_s^m the instant the preparation of message m starts. The instant the receive computation is finished is denoted as t_r^m.

In our system model we say that message m is in transit during the real-time interval $[t_s^m, t_r^m)$. We denote by $\delta^m = t_r^m - t_s^m$ the end-to-end computational + transmission delay of message m sent from one correct process to another. Further let $\mathcal{M}(t)$ be the set of all messages between correct processes that are in transit at real-time t. Let $\tau^-(t)$ be a lower envelope function on transmission delays of all messages that are in transit at real-time t, such that for any time t it holds that $\tau^-(t) \leq min(\delta^m)$ for all $m \in \mathcal{M}(t)$ if $|\mathcal{M}(t)| > 0$ and $\tau^-(t) = 1$ otherwise. We define for a fixed $\Theta \in \mathbb{R}, \Theta \geq 1$ the upper

[5] Such behavior can usually be simulated in the presence of crash faults by reliable broadcast. This is discussed later in Section 6.3.2 and Section 6.4.2.

envelope function $\tau^+(t) \leq \Theta\tau^-(t)$. At any time t it must hold that $\tau^+(t) \geq max(\delta^m)$ for all $m \in \mathcal{M}(t)$ if $|\mathcal{M}(t)| > 0$.

An example execution of some algorithm in the system model is depicted in Figure 3.1. At the top of the figure the message pattern is given. Below the two tightest bounds $\tau^+(t)$ and $\tau^-(t)$ are sketched in a diagram. This diagram is derived by constructing a square for every message m between t_s^m and t_r^m. At every instance the shortest message in transit defines $\tau^-(t)$ and the longest one $\tau^+(t)$. As in the example in Figure 3.1 the same message may define both τ^- and τ^+ at different times, depending on the other messages which are currently in transit.

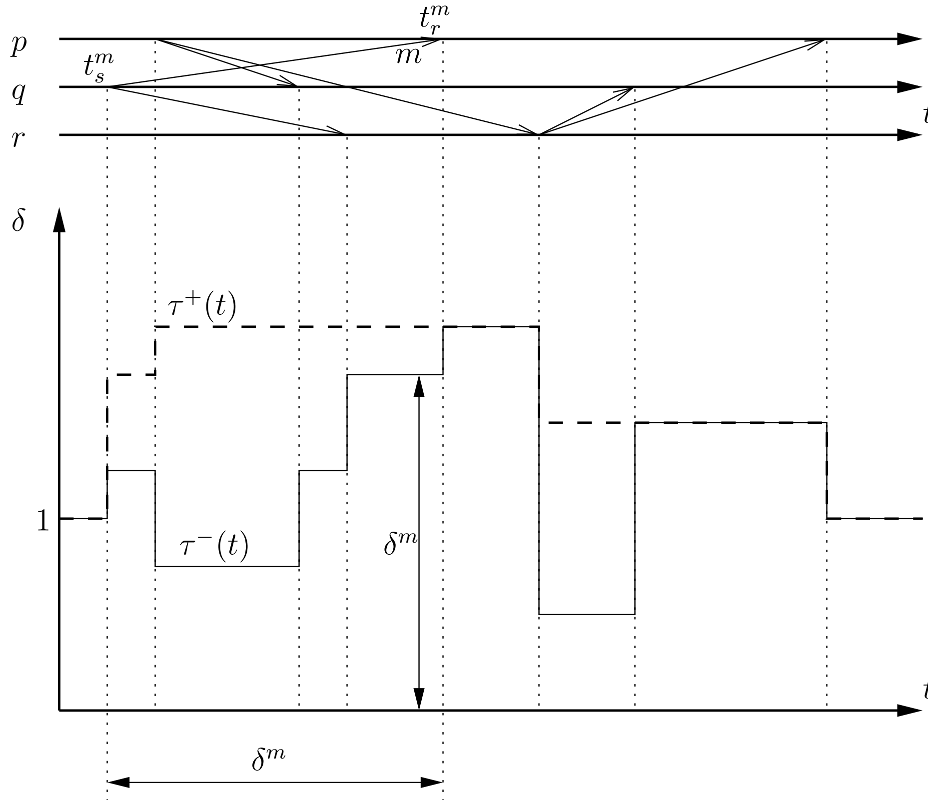


Figure 3.1. Timing of the System Model

3.3 Computational Model

We will denote by δ_{pq} the end-to-end computational + transmission delay of a message sent between two correct processes p and q; δ_{pq} can be different for each message.

Our computational model stipulates an upper bound τ^+ for the transmission delay as well as a lower bound τ^- such that $0 < \tau^- \leq \delta \leq \tau^+ < \infty$, where τ^- and τ^+ are *fixed but not known in advance.* Since $\tau^+ < \infty$, every message sent from a correct

process to another one is eventually received. Measures for the timing uncertainty are the *transmission delay uncertainty* $\varepsilon = \tau^+ - \tau^-$ and the *transmission delay ratio* $\Theta = \tau^+/\tau^-$.

3.4 Equivalence of the Models

We now show the equivalence of our two models. We argue that an observer of the system which is equipped with a "real-time clock" whose timebase emerges from the system is not able to distinguish executions in the system model from executions in the computational model.

Theorem 1 (Equivalence). *The system model and the computational model have the same expressive power.*

Proof. Let us assume an omniscient observer in both systems who is equipped with a clock which provides him with *observer-real-time*, i.e. a time base that look to him like real-time. More specifically, we assume that the observer-real-time t' is constructed as a function of Newtonian time t in the following way:

$$t' = \beta(t) = \int \frac{1}{\tau^-(t)}\, dt. \tag{3.1}$$

In the computational model, where $\tau^-(t) = \tau^-$, we hence get as time base $\beta(t) = t/\tau^- + c$, thus all correct message transmission delays take between 1 and Θ of observer-real-time here.

We now consider a message m between two correct processes in the system model. In order to get the measured value of the end-to-end delay we have to consider $\beta(t_r^m) - \beta(t_s^m)$. For any time $t \in [t_s^m, t_r^m)$, we have $t_r^m - t_s^m \geq \tau^-(t) \geq \frac{t_r^m - t_s^m}{\Theta}$. By monotonicity of integrals we get

$$1 \leq \beta(t_r^m) - \beta(t_s^m) = \int_{t_s^m}^{t_r^m} \frac{dt}{\tau^-(t)} \leq (t_r^m - t_s^m) \cdot \frac{\Theta}{t_r^m - t_s^m} = \Theta.$$

Since both, the system and the computational model are time free, no action can occur based on a timeout of a hardware clock etc. but only upon reception of a message and upon the completion of processor startup:

- *Processor startup.* Since processes start at unpredictable times, the occurrences of the very first action of every processor are completely unrelated in either model.
- *Message reception.* Apart from the very first action, all subsequent actions and hence messages sent by a correct process are direct responses to a received message.

When monitoring the execution of a distributed algorithm in system S using the appropriate observer-real-time clock, the observer cannot decide whether S adheres to the computational or to the system model. □

More informally one can argue about the equivalence when referring to the length of message chains: During the transmission of a (slow) message m, there cannot be a concurrent causal message chain which has at least one (broadcast or receive) event in common with m such that the chain consists of more than Θ messages during $[t_s^m, t_r^m)$.

Theorem 1 reveals that the system model which does not incorporate upper bounds on message transmission times (given that Θ holds) can in fact be reduced to a partially synchronous model with fixed but unknown upper and lower time bounds. It is hence possible to analyze distributed algorithms in the simple to handle computational model and transfer the results regarding safety and liveness to the more "elastic" system model.

Stating timeliness results in the context of real-time systems also requires special care: If the analysis in the computational model shows that some algorithm has a running time depending on τ^+, one has to be aware of the fact that these results must be translated to the system model where we just have the function $\tau^+(t)$. The underlying system's timing behavior during some execution of our algorithm hence emerges to upper network layers. Assume t_1 is the time when an execution starts according to observer-real-time base β, and t_2 when it ends. The elapsed real-time is then determined by $\Delta t = \beta^{-1}(t_2) - \beta^{-1}(t_1)$. When proving worst case response times in real-time systems, however, the strict upper bound on $\tau^+(t)$ of the considered system must be considered during the late binding process. Only by carefully derived upper bounds on message delays—by referring to the distributed real-time scheduling problem— timeliness properties can be said to be properly achieved.

Remark The definition of $\tau^-(t) = 1$ in the case of no messages in transit was introduced in order to construct $\beta(t)$ in the proof of Theorem 1 as an one-to-one mapping. It has no influence on bounds on durations of executions, and is only required during initialization of the algorithm (since we have no simultaneous start assumption). After initialization there will always be messages in transit and our envelope functions are defined in the natural way. One could also think of an extension of the Θ-Model that allows local timeouts (e.g. to reduce traffic). Such timeouts must properly taken care of when analyzing worst case response times, however.

3.5 Round Based Significant Uncertainty Ratio

In this thesis we will only consider round based algorithms in the Θ-Model. Such algorithms are executed in asynchronous rounds, i.e. every correct process sends a message for every round k. The transition to round $k+1$ occurs when $n-f$ messages for the current round are received. It will turn out that the uncertainty we have to deal with does not stem from the ratio of message delays directly but rather from the ratio of the longest message delay and the shortest round switching intervals. The shortest round-switching interval τ_f, however, is not determined by only a single correct message. Rather it is determined by the sending time of the first message and the receive time of the $n - f^{th}$ message. In the worst case this is irrelevant since any message is bounded by τ^-, and all could be sent simultaneously. From a practical point of view this is

very important, however. If one tries to establish an analytical expression for τ^- one would examine an idle system and the sending of a single message in this system—which could be a self reception as well. Obviously the receiver just has to deliver one message here. However, assuming that the receiver can process, say $n-f$ messages as fast as a single one is typically not valid in real systems. Choosing $\tau_f = \tau^-$ hence would be overly conservative since round switching requires $n-f$ messages, i.e. is determined by the $n-f$ fastest message. In particular, in broadcast bus networks one cannot transmit two messages simultaneously over the bus, i.e. the $n-f$ fastest messages must be transmitted one after the other. The time for $n-f$ messages to be transmitted in such networks is hence always larger than the best case time of sending a single message in an idle system. Using τ^- in the analysis would lead to over-valuations of the significance of τ^-.

Since we will consider several types of faults and hence different clock synchronization algorithms, the uncertainty ratio Θ used in our analysis will be slightly different for different fault classes.

3.5.1 Byzantine Faults

In our round based algorithms, the transition to the next round occurs when $n-f$ messages for the current round have been received i.e. when at least $n-2f \geq f+1$ messages sent by correct processes have arrived. In our timing analysis, we will hence set τ_f equal to the transmission time of the $n-2f$ fastest correct message. More specifically, we will use τ_f as expression for the shortest time it may take to send $n-f$ messages from distinct processes to one receiver (end-to-end). This choice is advantageous since τ_f need not be satisfied by all messages, but just by the $n-2f$ fastest one between two correct processes. Consequently, τ^- will not show up in the analytical expression for shortest round switching intervals explicitly, although of course $\tau_f \geq \tau^-$. The system's timing is therefore not violated even when up to f processes commit early timing failures, i.e. their messages are faster than τ_f. Let us formalize this observation:

Definition 1 (Incoming Messages). *For any correct process q, τ_q^- is the $n-2f$ smallest δ_{pq} for all messages sent by correct processes p that enable an event at q. τ_f is defined as the the smallest τ_q^- of all correct processes q.*

Lemma 1 (Sending Time). *If a correct process receives messages from at least $n-2f$ distinct correct processes by time t, then at least one message was sent by time $t-\tau_f$.*

Proof. Follows from Definition 1. □

Definition 2 (Significant Uncertainty Ratio). *The smallest feasible value of the uncertainty ratio Θ is $\Omega = \tau^+/\tau_f$.*

Remark Note that τ_f determines the time required for $n-f$ messages to be received by a correct process. There are two possible approaches here. Assuming $\tau_f > \tau^-$ leads to small values for Θ, and hence improved performance. This requires, however, that

all correct processes behave as expected regarding timing. Another approach, however, would consist of improving reliability, i.e. coverage. Setting $\tau_f = \tau^-$ we would allow up to f Byzantine faults and, additionally, f otherwise correct processes to commit early timing faults without violating the bound on the round-switching intervals.

3.5.2 Crash Faults

Asynchronous execution where just benign faults are considered are slightly different. We assume that processes are correct until they crash. Round switches will happen based on the reception of $n - f$ messages from correct (not yet crashed) processes. Hence τ_f can be defined similar to the Byzantine case by replacing $n - 2f$ with $n - f$ in Definition 1 and hence Lemma 1. Note carefully, however, that early timing faults must not happen here!

3.5.3 Clean-Crash Faults

Round switches will happen based upon the reception of message from $n - f \geq 1$ processes, thus $\tau_f = \tau^-$ must be considered. In many applications, however, f is considerably smaller than n. In such cases τ_f can be defined as in the crash fault case.

3.6 Related Work

The need for thinking about various types of synchrony started with the fundamental paper by Fischer, Lynch and Paterson [42] who showed that there is no deterministic algorithm for asynchronous systems that solves consensus if just one process may crash.

Dolev, Dwork and Stockmeyer [29] investigated how much synchronism has to be added to asynchronous systems in order to solve consensus. Dwork, Lynch and Stockmeyer [37] gave consensus algorithms for several kinds of partial synchrony. These results are also presented in the book by Lynch [71]. More work on agreement under diverse partially synchronous system models can be found in [8, 7, 81].

Chandra and Toueg [19] introduced the concept of unreliable *failure detectors* (FDs). Chandra, Hadzilacos, and Toueg [18] have shown the weakest type of unreliable FDs that solves consensus. More related work on FDs can be found in Section 6.5.

Timed semantics and the failure detector approach are compared by Charron-Bost, Guerraoui and Schiper [21] who show that synchronous systems are strictly stronger than asynchronous systems equipped with the perfect failure detector $\mathcal{P}$.

In real-time systems research, however, asynchronous algorithms have almost never been used. This is a consequence of the widespread believe that in order to build real-time applications timed algorithms have to be used [24, 100]. The design immersion principle (also referred to as late binding) by Le Lann [65, 66] has shown that this believe is wrong! Hermant and Le Lann [51] have shown that it is possible to use asynchronous algorithms efficiently to solve real-time computing problems. Moreover it is possible to show that implementations using late binding reach higher coverage. Regarding soft real-time systems there exists some work as well that deals with the usage of asynchronous algorithms [53].

Chapter 4

The Θ-Model Compared to other System Models

Having introduced our Θ-Model in Section 3, we will now investigate the differences with respect to classic approaches on modeling the timing of distributed systems between the two extrema lock-step synchrony and pure asynchrony. We consider here partial synchrony [37], semi-synchrony [81], and the Archimedean assumption [103].

4.1 Partial Synchrony

In the Θ-Model we consider end-to-end delays, i.e. $\tau^+(t)$ and $\tau^-(t)$, whereas partially synchronous—denoted as ParSync—models [37] incorporate delays as follows:

Δ is the upper bound on message delays (not end-to-end)

Φ is the upper bound on the relative computational speeds.

When taking a closer look at the definition of Φ one finds that the time base *ParSync-real-time* is defined as the frequency of steps of the fastest process. Therefore at any instant in ParSync-real-time a process may take at most one step. Furthermore, any correct process must take at least one step when the fastest one makes Φ steps. This has two important implications: From the theoretical point of view, ParSync turns out to be a special case of the Θ-Model, i.e. the Θ-Model is strictly closer to asynchrony in the hierarchy than ParSync. From the point of view of coverage, the ParSync model turns out to have limited abilities to model real systems.

Lets start with the theoretical point of view. Bounded Φ means that no computational step can be taken in 0 time. Regarding end-to-end delays (which are modeled in the Θ-Model) ParSync models must have $\tau^- > 0$: At any time a process may take at most one step where sending and receiving of some message cannot happen simultaneously. However, any end-to-end delay must be measured in ParSync as the interval between "the time process p sends a message to q" and "the time q sends its next message" (in reaction to p's). Since receive and send cannot happen at q at the same time we get $\tau^- \geq 1$.

Bounded Φ also means that there exists a finite lower bound on process speeds, i.e. no process is infinitely slow. In conjunction with the bounded message delay this implies a bounded end-to-end delay τ^+ as well.

To make a fair comparison of the synchrony assumptions of the Θ-Model and ParSync, we have to examine the models more closely. In [37] ParSync comes in two flavors:

ParSync.GST - Δ is known, and holds from some global stabilization time on.

ParSync.Unknown - Δ is unknown, and holds always.

From the existence of $\tau^- > 0$ and $\tau^+ < \infty$ we arrive at the conclusion that the Θ-Model and the ParSync model can be strictly ordered in hierarchy. For this we have to consider two additional versions of the Θ-Model (note that, in this thesis, we will mostly stick to the 'classic' Θ-Model when devising algorithms, since we are interested in algorithms suitable for distributed real-time systems. Nevertheless, we will refer to the other variants several times):

Θ.GST - some known Θ holds just from some global stabilization time on: We can employ one of the algorithms presented later (Theorem 11) which implements $\diamond\mathcal{P}$. It follows that ParSync.GST is a special case of this model with fixed lower and upper bounds on end-to-end delays that hold after GST.

Θ.Unknown - some unknown Θ always holds: We can update the timeout integer Ξ in the FD implementation (compare Section 6.2) whenever a message from a suspected process drops in. Eventually Ξ will become the correct value. It follows that ParSync.Unknown is a special case of this model with fixed lower and upper bounds on end-to-end delays.

These two findings imply that our algorithms could be run in ParSync models and would work as expected. Therefore the Θ-Model fully includes ParSync and is hence strictly closer to asynchrony.

As far as coverage of ParSync in real systems is concerned, we note that even the slowest process p must be able to receive a subset of the messages which were added to its message buffer since its last step. The synchrony assumption, however, requires that it must receive a message m at the latest Δ ParSync-real-time steps after m was put into its in buffer. Lets construct a worst case scenario in this model: Let $n-1$ processes be fast and send, at every step, a message to the single slow process p. If one wants to implement the model in a real system, the duration of a step must be such that p can deliver $(n-1)\Phi$ messages during a step. Otherwise the in-buffer of p could not be emptied at each step, such that it would require unbounded memory.

We believe that this modeling has some shortcomings: Fast and slow processes require the same amount of time for a step. When looking at queuing issues, it becomes obvious that the length of a step must depend on the time a slow process requires to deliver $(n-1)\Phi$ messages. Therefore the performance of a fast process is limited

by the delivering speed of a slow one. Consequently, there exist a considerably large $\tau^- = 1_{ParSync}$ in ParSync models, which is far away from being 0.

It follows that ParSync's definition of computation speed has influence only on sending messages. For receiving messages a slow process and a fast one deliver the same number of messages during the same time interval. Just that a fast process delivers some messages every instant and the slow one delivers many messages just every Φ instants. This is not a proper way of modeling slow process and clearly inferior to the modeling of end-to-end delays in the Θ-Model.

4.2 Semi-Synchrony

The usage of the term partial synchrony [37, 29] as described in Section 4.1 has slightly changed in recent years. In literature models are called ParSync now which where originally referred to as semi-synchronous (SemiSync), obviously since in SemiSync [80, 81, 8] there also exist Φ and Δ (compare Section 4.1). There is, however, a slight difference: Since processes may deliver messages and send responses in zero time, there does not exist some bounded $\tau^- > 0$ in this model. The absence of this synchrony-enabling parameter is compensated by referring to bounded-drift local clocks (timers, hardware clocks, watchdogs etc.). The existence of some (possibly unknown) Φ and Δ (and hence τ^+) makes it possible to detect crashes based on approximate local information on the progress of real-time provided by local clocks.

We will later see in Section 5.7 that even the existence of local clocks does not help (w.r.t. increasing performance or number of solvable problems) in the Θ-Model. This is due to the fact that $\tau^+(t)$ does not imply a strict upper bound on end-to-end delays. We now informally discuss this property by referring to SemiSync.

The way how clocks are used in SemiSync relies on the following property: Due to bounded Φ and Δ and the existence of a bounded drift clock, a correct process's clock ticks at most some x times until some message from a correct process must arrive. In the Θ-Model, Φ and Δ—represented now by $\tau^+(t)$—vary over time, such that bounded drift clocks are of no help.

We believe that SemiSync models do not exploit properly the inherent synchrony of all real computer networks: Setting the lower bound to 0 is obviously a safe assumption, but real systems have lower bounds on message transmissions (due to the physical impossibility of transmitting and processing information in zero time).

From a theoretical viewpoint SemiSync is of course of highest interest since its lower bounds on termination times apply to weaker models as well.

Non-technical Remark Before returning to more technical topics, lets have a look at the aesthetics of the Θ-Model when compared SemiSync (and to ParSync): In SemiSync, times from different domains are compared: Message delays and clock frequencies. We believe that these two domains are totally independent, such that comparing them, in order to timeout processes is not an elegant solution. (Whether elegance should be left in the domain of tailors and shoemakers will not be discussed here.) Exploiting the inherent synchrony of the system (recall $\tau^- > 0$ in any real system) and comparing (somehow) slow and fast messages is preferable from this viewpoint as well.

4.3 The Archimedean Assumption

The Archimedean assumption was introduced by Vitányi [102, 103] and limits a distributed systems asynchrony. It was employed in the context of message and time complexity bounds for distributed systems with realistic timing behavior. The same assumption was employed by Spirakis and Tampakas [94] for the same purpose. Some remarks on this work can also be found in the book by Tel [98].

The model based upon the Archimedean assumption involves

- an a priori known upper bound u on message transmission delay + time interval between two process steps (respectively clock ticks in [103])
- a lower bound m on the time between two steps of a process (resp. ticks of local clocks)
- some bounded $s \geq u/m$.

Vitányi [103] uses this synchrony assumption to show that time-driven algorithms can be devised that achieve smaller message and/or time complexity compared to algorithms in purely asynchronous systems. In [103] time-driven algorithms for leader election (see also [102]), spanning tree construction, and (hardware) clock synchronization are given. Spirakis and Tampakas [94] provide timed algorithms for mutual exclusion, symmetry breaking in a logical ring and the problem of readers and writers.

Although it seems that the synchrony assumptions of the Θ-Model and the Archimedean assumption are similar, it must be noted that there is a subtle difference. Unlike the lower bound τ^- in the Θ-Model, the lower bound m in the Archimedean model does not involve message reception. Hence, whereas τ^- and τ^+ are likely to be correlated via queuing delays such that Θ may hold even in case of high/overload, there cannot be any correlation between m and u in case of increasing transmission delays. The coverage of Θ is hence higher than those of s.

The Archimedean assumption differs from the Θ-Model also in the fact that the former's synchrony is primarily used for simulating local "hardware clocks" by just executing computing steps in a spin loop. Due to the definition of s, this clock can be used to timeout messages. The same could of course be done in the Θ-Model, by doing self-reception in a spin loop. Note that this would even work when transmission delays go up in periods of overload, since self reception is also done via the queues of a processor (rather than just writing into memory). Still, the algorithms employed in this thesis do not use this technique, which has the additional disadvantage that it would not allow us to utilize the typically larger value of τ_f (cp. Section 3.5).

Last but the not least, the problems addressed in this thesis are very different from the problems investigated under the Archimedean assumption. Like Le Lann and Schmid [67], we employ the Θ-Model as a weak partially synchronous system model, suitable for solving consensus and related problems in a time free manner. By contrast, the work in [103, 94] is basically concerned with reducing complexity by adding synchrony assumptions to the system.

Chapter 5

Implementation Issues

5.1 Design Immersion

This section describes the work that has to be done when immersing solutions of this thesis into real systems in order to achieve real-time properties. Our results have the following property in common: "If we know values for Θ and $\tau^+(t)$ during the execution of some algorithm we can derive the termination times". In order to get bounded termination times we hence require bounds for Θ and $\tau^+(t)$. The bound $\bar{\tau}^+ \geq \tau^+(t)$ for all times t is hence an important factor (although it is not compiled into our algorithms). It is the solution to the worst case response time associated with the distributed real-time scheduling problem. Careful analysis for the targeted system has to be done to derive this value. This analysis [51] depends on the type of the network and is out of the scope of this thesis.

Similarly, for all times t, $\bar{\tau}^- \leq \tau^-(t)$ is the lower bound and hence the best case solution to the distributed real-time scheduling problem. It would be perfectly safe to use the ratio $\bar{\Theta} = \bar{\tau}^+/\bar{\tau}^-$ in order to derive a value for Θ. This approach, however, would be very inefficient (Θ will show up in our termination time bounds, and it is the only value that has to be known a priori to the algorithms). This is where the system model is required: In most systems in which distributed algorithms are executed one cannot have message end-to-end delays of $\bar{\tau}^+$ and $\bar{\tau}^-$ at the same time. In high load periods (e.g. contention on a shared medium) some message transmissions will require $\bar{\tau}^+$. But no message in transit will require $\bar{\tau}^-$ during this periods. This can be used to decrease the actual value of Θ by referring to the system model where $\tau^+(t)$ and $\tau^-(t)$ are variant: We are faced with a variant of the distributed real-time scheduling problem: How are $\tau^+(t)$ and $\tau^-(t)$ related in the target system? In many systems the solution to this problem generates a bound on Θ which is considerably smaller than $\bar{\Theta}$.

5.2 Assumption Coverage

This section presents several arguments that should strengthen the strong connection of real systems and the Θ-Model. We do so by reviewing some arguments [67] that should be considered when reasoning about the coverage of the Θ-Model.

Since the Θ-Model is fundamentally different from other well known partially synchronous models [37, 19] the arguments regarding coverage must obviously be slightly different. It is important to notice that the Θ-Model does not stipulate an upper bound Δ upon message transmission times (see the system model in Section 3.2). The synchrony assumption is rather stipulated explicitly by establishing a relation of longest and shortest message transmission delays. We hence must compare the coverage of the Δ-assumption to the coverage of the Θ-assumption. We consider here the "pessimistic" approach from Section 5.1, i.e. we consider strict upper and lower bounds $\bar{\tau}^+$ and $\bar{\tau}^-$, respectively such that $\bar{\Theta} = \bar{\tau}^+/\bar{\tau}^-$. This approach is pessimistic in the sense that it gives the largest value for Θ, while it reaches higher coverage as discussed in the following.

Let a model with an assumption on Δ (and hence τ^+) have a value for coverage c_Δ. A model which assumes a bound on $\bar{\Theta}$ has a value for coverage $c_\Theta \geq c_\Delta c_{\tau^-}$. Since τ^- is the solution of the best case schedulability analysis—which should be very simple to derive compared to worst case analysis—its coverage can be considered 1. Hence c_Θ is at least as good as c_Δ. If we can find just one execution where the Θ-assumption holds whereas the Δ-assumption is violated we have shown that the coverage of Θ is in fact better. That such executions exists is obvious from queuing issues which were already discussed in Section 2.1.

5.3 Real-Time Scheduling

It should be clear by now that real-time guarantees do not come for free. Considerable work by computer scientists is required in order to derive the required figures for $\bar{\tau}^+$, $\bar{\tau}^-$, and Θ by analyzing distributed scheduling algorithms. We now give some numbers on these bound which can be found in real-time literature:

When we regard τ^+ and τ^- as clock synchronization (resp. failure detector) level end-to-end delays [51], the resulting delay uncertainty ε is usually much smaller than the uncertainty $\varepsilon_A = \tau_A^+ - \tau_A^-$ at application-level. Typical values for the latter are $\tau_A^- = 100\mu$s and $\tau_A^+ = 10\ldots100$ms, which would lead to some $\Theta_A = 100\ldots1000$. Rather, τ^+ and τ^- are the worst case and best case response times, respectively, associated with the distributed real-time scheduling problem underlying the distributed clock synchronization execution. Typical values for Θ reported in real-time systems research [39, 49] are $5\ldots10$.

5.4 Experimental Results

In order to get an idea of the value of Θ some experiments were conducted by Albeseder [5]: The basic clock synchronization algorithm from Section 6.1 was executed on Linux workstations connected by switched Ethernet and a custom monitoring software was used to determine the values of the message end-to-end delays. The clock synchronization algorithm from Section 6.1 was run like a fast failure detector using high-priority threads and head-of-the-line scheduling [51]. During a run, varying network as well as processor load in the range of 1..60% was artificially induced. Similar

behavior of the network was observed in all 29 runs with a length of 120s each. The numbers we report here are average values over all runs: Absolute bounds on message end-to-end delays were $\bar{\tau}^- \approx 23\mu$s and $\bar{\tau}^+ \approx 640\mu$s. Pessimistically this would lead to $\bar{\Theta} \approx 27.8$. The effective Ω, however, was measured according to our system model and thus considered only those messages that are simultaneously in transit. Moreover, following Section 3.5, it is not the duration of one fastest message τ^- that determines the behavior of our algorithms but just the duration τ_f of the $n-2f$th fastest message from a correct process. Obviously, this makes a difference in real networks since the usually small self reception time has no influence on the overall timing. This has been confirmed by the experiments, which revealed a value of $\Omega \approx 9.8$.

Those and other results are strong evidence that $\tau^-(t)$ and $\tau^+(t)$ are indeed correlated, i.e. all end-to-end delays of all messages grow in high load situations. Moreover, these results showed that even in these settings—which are far from being real-time networks that deliver much less timing uncertainty—the uncertainty ratio Θ remains within reasonable bounds.

5.5 Deterministic Ethernet

In [52] it was shown how an implementation of the perfect failure detector $\mathcal{P}$ in the Θ-Model can be immersed into a real system. The work rests upon an architecture built atop Deterministic Ethernet (which has various nice properties, e.g. upper bounds on message delays). This architecture followed the fast FD approach [51] where failure detection is done low level and higher level application can make use of it (similar to clock synchronization, where the clocks are synchronized low level in order to reduce the uncertainty). For this architecture it was shown that the time free solution can be implemented efficiently by using suitable queuing and bus arbitration schemes: It has been shown analytically that the architecture build atop of deterministic Ethernet delivers a timing behavior characterized by $\Theta = 1$. This approach hence allows to implement totally time free solutions for consensus in real networks, without sacrificing real-time behavior.

5.6 Algorithms at Application Level

The values for Θ we discussed in Section 5.3, Section 5.4, and Section 5.5 are mostly in the context of implementations of the fundamental clock synchronization algorithms (cp. Section 6.1) respectively failure detectors (cp Section 6.2). In order to reduce uncertainty they will typically be implemented at low level such that performance is comparable to fast failure detectors [51].

Other algorithms in this thesis—like those presented e.g. in Section 6.3 or Section 6.4—will typically be executed at application level such that values for Θ at this level must be derived analytically as well. These values may me considerably larger than those discussed in the previous sections, however.

5.7 Bounded Drift Clocks

Most of the computers which are considered for building reliable real-time system are equipped with hardware clocks. The question arises how such clocks could be employed in order to improve the system performance, reduce overhead etc.

Let's consider a variant of the Θ-Model where processes are equipped with good clocks, i.e. clocks with bounded drift. Models like this where the synchrony assumption is a strict upper bound on message delays are called SemiSync (cp. Section 4.2), where usually faults are detected by setting timeouts. We do not find any improvement by such an approach under the Θ–assumption. First, $\tau^+(t)$ has no upper bound, such that the system model is too weak to timeout processes using local hardware clocks by just measuring one round trip and estimating $\tau^+(t)$ then. This is because $\tau^+(t)$ can unboundedly increase in future. But even if not just one round trip, but $\lfloor \Theta + 1 \rfloor$ causally related round trips to some process p are measured, such that these must overlap with one round trip to some process q, there is no gain in performance. It can easily be shown by example that this solution, when immersed into real systems (cp. Section 5.1) may have worse termination times than our approach (see Section 6.2), while reducing coverage by referring to timed semantics. Moreover such measurement has to be repeated for every timeout.

Local clocks (in fact any approximate local notion of elapsed time) can be employed, however, for controlling the overhead of our algorithms. Consider the failure detector in Section 6.2 that always requires messages to be in transit. If this would be implemented like this, there would not be any bandwidth left for other algorithms that use the failure detector. To make this algorithm suitable for real systems, local timeouts are used just to keep the failure detector quiet for some time [52, 68]. Note that even if we used clocks in such a way, we would still have time free algorithms, in the sense that the clocks are not used to measure remote events of message delays. Hence, the correctness of the algorithms do not rely on an assumed relation of clock tick intervals and message delays.

Chapter 6

Selected Algorithms in the Θ-Model

This section introduces fault-tolerant algorithms for the Θ-Model. Here, we consider the simple case where all processes are initially up and listening to each other. This assumption will be dropped in Section 7 and Section 8, where we consider system booting in detail. We start with clock synchronization which is the primitive underlying all our other algorithms. We present an implementation of the perfect failure detector $\mathcal{P}$ for solving consensus, an implementation of atomic broadcast that uses our clocks directly instead of referring to $\mathcal{P}$, and some non-blocking atomic commitment protocols.

From a theoretical viewpoint Section 6.3 is is the most surprising one since it gives a solution to the strongly dependent decision problem. This reveals that the Θ-Model is as powerful as a synchronous model although it stipulates no upper bound on message transmission delays in the system model. We further give an algorithm that simulates lock-step synchrony, which makes it possible to execute any synchronous algorithm in a totally time free way—even if Byzantine faults are considered.

6.1 Clock Synchronization

Clock synchronization is a fundamental service in many distributed systems. Although traditionally studied in systems with a priori known timing behavior where all the processes are equipped with hardware clocks, it can also be solved in partially synchronous systems with software clocks (counters). Formally the problem of clock synchronization reads as follows.

Definition 3 (Clock Synchronization). *Every correct process p maintains an integer-valued clock $C_p(t)$, that can be read at arbitrary real-times t. It must satisfy:*

(π) Precision Requirement. *There is some constant precision $D_{max} > 0$ such that $|C_p(t) - C_q(t)| \leq D_{max}$ for any two correct processes p and q and any real-time t.*

(α) Accuracy Requirement. *There are some constants $a, b, c, d > 0$ such that $a(t_2 - t_1) - b \leq C_p(t_2) - C_p(t_1) \leq c(t_2 - t_1) + d$ for any correct process p and any two real-times $t_2 \geq t_1$.*

According to (π) the difference of any two correct clocks in the system must always be bounded, whereas (α) guarantees some relation between progress of clock-time and progress of real-time. In literature (α) is often referenced as *linear envelope requirement*. The constants a, b, c, and d determine how fast logical time (measured in ticks representing some logical time unit) progresses with respect to real-time. In case of our solution, a clock ticks whenever its clock synchronization algorithm enters the next round of computation.

6.1.1 Byzantine Case

The algorithm given in Figure 6.1 is a variant of the classic non-authenticated clock synchronization algorithm by Srikanth and Toueg [95]. It provides the fundamental properties (π) and (α) in our Θ-Model in systems with at most $f < n/3$ Byzantine faults. It is started by sending *(round* 0*)* in `line 3`. If a correct process receives $f + 1$ *(round l)* messages (see `line 5`) it can be sure that at least one of those was sent by a correct process and therefore also sends *(round l)*. If a process receives $n - f \geq 2f + 1$ *(round k)* messages (`line 10`) it can be sure that at least $f + 1$ of those will be received by every correct process—which then executes `line 5`—, such that within bounded time all correct processes receive $n - f$ *(round k)* messages. Therefore when receiving these $n - f$ *(round k)* messages a correct process updates its clock to $k + 1$. We will see in the following analysis that this behavior suffices to guarantee (π) and (α).

```
0:  VAR k : integer := 0;
1:
2:  /* Initialization */
3:  send (round 0) to all [once];
4:
5:  if received (round ℓ) from at least f + 1 distinct processes with ℓ ≥ k
6:     → k := ℓ; /* jump to new round */
7:        send (round k) to all [once];
8:  fi
9:
10: if received (round k) from at least n − f distinct processes
11:    → k := k + 1;
12:       send (round k) to all [once]; /* start next round */
13: fi
```

Figure 6.1. Clock Synchronization Algorithm

The algorithm given in Figure 6.1 is totally time free, i.e. it has no knowledge of τ^+, τ^-, ε, and not even Θ. Its properties regarding (π) and (α) solely emerge from the underlying system's timing properties. Therefore, violations of any assumptions of the system timing can do no harm to the algorithm per se. It is just the properties (π) and (α) that get out of the derived bounds. If the system returns to expected behavior (π) and (α) return to their expected values as well[1]. In the algorithm given in Figure 6.1

[1]In the implementation of the eventually perfect failure detector $\diamond\mathcal{P}$ this property is of major

the value k is the current clock value of the process. In the analysis we will often refer to it by $C_p(t)$. We start the analysis with some definitions.

Definition 4 (Local Clock Value). *$C_p(t)$ denotes the local clock value of a correct process p at real-time t; σ_p^k, where $k \geq 0$, is the sequence of real-times when process p sets its local clock value to $k+1$.*

Definition 5 (Maximum Local Clock Value). *$C_{max}(t)$ denotes the maximum of all local clock values of correct processes at real-time t. Further, let $\sigma_{first}^k = \sigma_p^k \leq t$ be the real-time when the first correct process p sets its local clock to $k+1 = C_{max}(t)$.*

We start our analysis with the following observation in Lemma 2, which is required in order to show our major properties later.

Lemma 2 (2nd if). *The first correct process that sets its clock to $k = C_{\max}(t) > 0$ by time t must do so by `line 10` and hence sends (round $C_{\max}(t)$) by time t. By t at least $n - 2f$ correct processes have clock value $k - 1$ and have sent their (round $k - 1$) message.*

Proof. By contradiction. Assume that the first correct process p sets its clock to $k = C_{\max}(t)$ at instant t by `line 5`. At least one correct process must have sent a message for a tick $l \geq k$ to enable the rule at p. Since correct processes only send messages for ticks less or equal their local clock value, at least one must already have had a clock value $l \geq k$ at instant t. This contradicts p to be the first one that reaches clock value k. When entering round k by `line 10` process p sends *(round $C_{\max}(t)$)*. □

The following Theorem 2 introduces the four basic properties of our algorithm. These properties[2] will be used in this thesis to derive all other results. They hence apply to any algorithm satisfying (P), (U), (Q), and (S).

Theorem 2 (Clock Synchronization Properties). *In systems of $n \geq 3f + 1$ processes, the algorithm given in Figure 6.1 achieves:*

(P) Progress. *If all correct processes set their clocks to k by time t, then every correct process sets its clock at least to $k + 1$ by time $t + \tau^+$.*

(U) Unforgeability. *If no correct process sets its clock to k by time t, then no correct process sets its clock to $k + 1$ by time $t + \tau_f$ or earlier.*

(Q) Quasi Simultaneity. *If some correct process sets its clock to k at time t, then every correct process sets its clock at least to $k - 1$ by time $t + \varepsilon$.*

(S) Simultaneity. *If some correct process sets its clock to k at time t, then every correct process sets its clock at least to k by time $t + \tau^+ + \varepsilon$.*

---

interest. Since our implementation can only deliver eventual semantics during booting, the consensus algorithm must live with this semantics anyway. Our implementation will hence always guarantee safety, even when timing assumptions are temporarily violated—compare Section 6.2.

[2] Anticipating the context of system booting [106, 109], it should be noted that (U) and (Q) hold for any number of participating processes—compare Section 7.

Proof. We show the properties separately.
Progress. By assumption at least $n-f$ correct processes set their clocks to k, and hence send *(round k)* by time t. These messages must be received by all correct processes by time $t+\tau^+$. Thus they set their clocks to $k+1$ by `line 10`, if they have not already done so.

Unforgeability. Assume by contradiction that a correct process p sets its clock to $k+1$ by time $t+\tau_f$. Correct processes may set their clock by (1) `line 10` or (2) `line 5`.

Assume (1). Process p does so based on $n-f$ *(round k)* messages by distinct processes, i.e. at least $n-2f$ were sent by correct processes. By Lemma 1 at least one of these messages was sent by time t, which provides the required contradiction.

Assume (2). Process p does so based on $f+1$ *(round $k+1$)* messages, i.e. at least one correct process has already clock value $k+1$ by time $t+\tau_f$. The first correct process which has set its clock to $k+1$ must have done so by `line 10` by Lemma 2. By applying (1) we again derive a contradiction.

Quasi Simultaneity. Let process p be the first correct process to set its clock to k. By Lemma 2 p sets its clock to k using `line 10`. This happens based on $n-f \geq 2f+1$ *(round $k-1$)* messages by distinct processes. At least $f+1$ of these must be sent by correct processes whose messages are received by all correct processes by $t+\varepsilon$. They then set their clocks to $k-1$ by `line 5`, if they have not already done so.

If process p is not the first process which sets its clock to k, then at least one correct processes has done so before at time $t' < t$. By the same reasoning as above all correct processes must set their clocks to $k-1$ by time $t'+\varepsilon < t+\varepsilon$.

Simultaneity. According to (Q) all correct processes set their clock to $k-1$ by $t+\varepsilon$. By (P) all processes set their clock to k by time $t+\tau^+ +\varepsilon$. □

6.1.2 Clock Synchronization Properties

The following analysis solely bases on the properties (P), (U), (Q), and (S). It may hence be applied to any clock synchronization algorithm that achieves these properties (e.g. the algorithms of the following sections, which achieve even uniform versions of our properties for restricted fault types). We now give some technical lemmas and necessary definitions in order to derive our theorems for precision (π).

Lemma 3 (Fastest Progress). *Let the first correct process set its clock to k at time t. Then no correct process can reach a larger clock value $k' > k$ before $t+\tau_f(k'-k)$.*

Proof. By induction on $l = k'-k$. For $l=1$, Lemma 3 is identical to (U) and therefore true. Assume that no correct process has set its clock to $k+l$ before $t+l\tau_f$ for some l. Thus no correct process may set its clock to $k+l+1$ before $t+l\tau_f+\tau_f = t+\tau_f(l+1)$ by (U). Hence Lemma 3 is true for $l+1$ as well. □

In our analysis we will frequently require a bound upon the maximum progress of $C_{\max}$ during a given real-time interval $[t_1, t_2]$. Lemma 3 can be applied for this purpose if $t_1 = \sigma^k_{first}$, but not if t_1 is arbitrary. As in [89], we unify those cases via the following Definition 6.

Definition 6 (Synchrony). *Real-time t is in synchrony with $C_{max}(t)$ iff $t = \sigma^k_{first}$ for some arbitrary k, as defined in Definition 5. Let the indicator function of non-synchrony be defined as*

$$I_{t \neq \sigma} = I_\sigma(t) = \begin{cases} 0 & \textit{if } t \textit{ is in synchrony with } C_{max}(t), \\ 1 & \textit{otherwise.} \end{cases}$$

Lemma 4 (Maximum Increase of C_{max} within Time Interval). *Given any two real-times $t_2 \geq t_1$, then $C_{max}(t_2) - C_{max}(t_1) \leq \lfloor \frac{t_2 - t_1}{\tau_f} \rfloor + I_\sigma(t_1)$.*

Proof. Let $k = C_{max}(t_1) - 1$. We have to distinguish the two cases $\sigma^k_{first} = t_1$ and $\sigma^k_{first} < t_1$.

Let $\sigma^k_{first} = t_1$ such that $I_\sigma(t_1) = 0$. From Lemma 3 follows that C_{max} may increase every τ_f time units, hence $\lfloor \frac{t_2 - t_1}{\tau_f} \rfloor$ times before t_2. Since $I_\sigma(t_1) = 0$, Lemma 4 is true for this case.

Now let $\sigma^k_{first} < t_1$, such that $I_\sigma(t_1) = 1$, and let the real-time $t' = \sigma^{k+1}_{first} > t_1$. We can now apply Lemma 3 starting from time t'. Since $t_2 - t_1 > t_2 - t'$ it follows from Lemma 3 that C_{max} cannot increase more often than $\lfloor \frac{t_2 - t_1}{\tau_f} \rfloor$ times between t' and t_2. At instant t', C_{max} increases by one such that $C_{max}(t_2) - C_{max}(t_1) \leq \lfloor \frac{t_2 - t_1}{\tau_f} \rfloor + 1$. Since $I_\sigma(t_1) = 1$ Lemma 4 is true. □

The fundamental properties (Q) and (S) just state that all correct processes set their clocks to some value within a bounded real-time interval. Precision can be derived from how often a fast process advances its clock within this interval. Since we have two different properties, we now derive two bounds on precision.

Theorem 3 (Precision by Quasi Simultaneity). *Any algorithm which guarantees (P), (U), and (Q) satisfies the precision requirement (π) with $D_q = \lfloor \Omega + 2 \rfloor$.*

Proof. If no correct process advances its clock beyond $k = 2$, precision $D_q \geq 2$ is automatically maintained since all clocks are initially synchronized to $k = 0$.

Assume that a correct process p has local clock value $k \geq 0$ within a still unknown precision D_q with respect to all other correct processes—and therefore also to $C_{max}(t')$—at real-time t'. We use (Q), Definition 6 and Lemma 4 to reason about D_q by calculating $C_{max}(t)$ for some time $t > t'$.

Let process p advance its clock to $k+1$ such that $\sigma^k_p = t > t'$. Since p has not done so before t, no other correct process has set its clock to $k+2$ before $t - \varepsilon$, following directly from (Q), thus $\sigma^{k+1}_{first} \geq t - \varepsilon$.

From Lemma 4 follows that $C_{max}(t) \leq \lfloor \frac{t-(t-\varepsilon)}{\tau_f} \rfloor + I_\sigma(t-\varepsilon) + C_{max}(t-\varepsilon)$. Let us now take a closer look at the term $I_\sigma(t-\varepsilon) + C_{max}(t-\varepsilon)$: If $\sigma^{k+1}_{first} = t - \varepsilon$ and therefore $t - \varepsilon$ is synchronized with C_{max}, $C_{max}(t-\varepsilon) = k+2$ and $I_\sigma(t-\varepsilon) = 0$ (following Definition 6). If on the other hand $\sigma^{k+1}_{first} > t - \varepsilon$, then $C_{max}(t-\varepsilon) = k+1$ and $I_\sigma(t-\varepsilon) = 1$. In both cases $I_\sigma(t-\varepsilon) + C_{max}(t-\varepsilon) = k+2$ such that $C_{max}(t) \leq \lfloor \frac{\varepsilon}{\tau_f} \rfloor + k + 2 = \lfloor \frac{\tau^+ - \tau^-}{\tau_f} + 2 \rfloor + k \leq \lfloor \Omega + 2 \rfloor + k$. Thus $C_{max}(t) \leq \lfloor \Omega + 2 \rfloor + k$.

Process p has clock value $C_p(t') = k$ at time $t' < t$ which is by assumption within precision. Since $C_p(t') < C_p(t)$ and $C_{max}(t') \leq C_{max}(t)$, we get a bound for D_q from the difference $C_{max}(t) - C_p(t') = C_{max}(t) - k \leq \lfloor \Omega + 2 \rfloor$. □

Theorem 4 (Precision by Simultaneity). *Any algorithm which guarantees (P), (U), (Q), and (S) satisfies the precision requirement (π) with* $D_s = \lfloor 2\Omega + 1 \rfloor$.

Proof. The same arguments as in the proof of Theorem 3 apply here as well. By employing times and values of (S) instead of (Q) the bound on precision can easily be derived. □

Both D_q and D_s are valid bounds for (π). In general, however, D_q is the tighter one.

Theorem 5 (Precision). *Any algorithm which guarantees (P), (U), (Q), and (S) achieves precision (π) with* $D_{max} = \min\{D_q, D_s\}$.

Proof. Both Theorem 3 and Theorem 4 apply. □

After our proof of precision we now start our analysis of accuracy with some technical lemmas.

Lemma 5 (Slowest Progress). *Let p be the last correct process that sets its clock to k at time t. No correct process can have a smaller clock value than $k' > k$ at time* $t + \tau^+(k' - k)$.

Proof. By induction on $l = k' - k$. For $l = 1$, Lemma 5 is identical to (P) from Theorem 2 and therefore true. Assume that the last correct process sets its clock to $k + l$ at time $t + \tau^+ l$ for some l. Note that all correct processes have set their clocks to $k + l$ by then. According to (P), every correct process must set its clock to $l + 1$ by time $t + \tau^+ l + \tau^+ = t + \tau^+(l + 1)$. Hence, Lemma 5 is true for $l + 1$ as well. □

Lemma 6. *Let p be a correct process that sets its clock to k at time t. Then, p sets its clock to $k + 1$ by time* $t + 2\tau^+ + \varepsilon$.

Proof. The proof uses the properties (P) and (S) from Theorem 2. Simultaneity (S) guarantees that all correct processes set their local clocks to k by time $t' = t + \tau^+ + \varepsilon$. Progress (P) guarantees that all correct processes, including p, must set their clocks to $k + 1$ by time $t' + \tau^+$. Hence p sets its clock by time $t + 2\tau^+ + \varepsilon$. □

Lemma 7 (Lower Envelope Bound). *Any algorithm which guarantees (P), (U), (Q), and (S) ensures that the clock of every correct process p satisfies* $\frac{t_2 - t_1}{\tau^+} - 5 + \frac{2}{\Theta} < C_p(t_2) - C_p(t_1)$ *for all times* $t_2 \geq t_1$.

Proof. Let $C_p(t_1) = k + 1$ and $C_p(t_2) = k + l + 1$ for some $l \geq 0$. Process p has set its clock to $k + 1$ at time $\sigma_p^k \leq t_1$ and to $k + l + 1$ at time $\sigma_p^{k+l} \leq t_2 < \sigma_p^{k+l+1}$. Hence, $t_2 - t_1 < \sigma_p^{k+l+1} - \sigma_p^k \leq \sigma_p^{k+l} - \sigma_p^k + 2\tau^+ + \varepsilon$ according to Lemma 6 and thus $t_2 - t_1 - 2\tau^+ - \varepsilon < \sigma_p^{k+l} - \sigma_p^k$. Using (S) in Theorem 2 in conjunction with Lemma 5, it follows that $\sigma_p^{k+l} - \sigma_p^k \leq l\tau^+ + \tau^+ + \varepsilon$ and hence $t_2 - t_1 - 2\tau^+ - \varepsilon < l\tau^+ + \tau^+ + \varepsilon$. Since $l = C_p(t_2) - C_p(t_1)$, our lower envelope bound evaluates to $\frac{t_2 - t_1}{\tau^+} - 5 + \frac{2}{\Theta} < C_p(t_2) - C_p(t_1)$ as asserted. □

Lemma 8 (Upper Envelope Bound). *Any algorithm which guarantees (P), (U), (Q), and (S) ensures that the clock of every correct process* p *satisfies* $C_p(t_2) - C_p(t_1) < \frac{t_2 - t_1}{\tau_f} + D_{max} + 1$ *for all times* $t_2 \geq t_1$.

Proof. From precision in Theorem 5 follows that $C_{\max}(t) - D_{max} \leq C_p(t) \leq C_{\max}(t)$ at all times t. Thus $C_p(t_2) - C_p(t_1) \leq C_{\max}(t_2) - C_{\max}(t_1) + D_{max}$. Applying Lemma 4, the statement of our lemma follows immediately. □

Theorem 6 (Accuracy). *Any algorithm which guarantees (P), (U), (Q), and (S) achieves accuracy* (α) *with*

$$\frac{t_2 - t_1}{\tau^+} - 5 + \frac{2}{\Theta} < C_p(t_2) - C_p(t_1) < \frac{t_2 - t_1}{\tau_f} + D_{max} + 1$$

for all correct processes p *and all times* $t_2 \geq t_1$.

Proof. Follows from Lemma 7 and Lemma 8. □

6.1.3 Restricted Failure Modes

As mentioned above, clock synchronization is the fundamental service in the Θ-Model. Since many problems in distributed computing (e.g. non-blocking atomic commitment) are studied in the presence of minor faults as clean-crash or crash faults only, we now give clock synchronization algorithms for such faults as well. Although it is possible to use the algorithm from the Byzantine case for this purpose, in applications it is nevertheless better to decrease the required number of processes to tolerate less severe faults.

This section's algorithms reach the same properties as given in Theorem 2. Therefore the derived properties (π) and (α) from Section 6.1.2 are achieved as well. This allows us to regard the clock synchronization layer as black box which provides identical properties regarding timing no matter what failure modes are considered.

In the following we will state properties that hold for all processes, uniformly. Of course we cannot guarantee any properties for processes that have crashed. When we argue about all processes we mean all processes that are still alive.

Clean-Crash Faults

By clean-crash faults we mean processes which remain correct until they stop operation. Any step in the algorithm is executed completely or not at all. The critical operation in this context of the algorithm given in Figure 6.2 is the send operation (send message to all): We assume that processes do not crash during the execution of an **if**-statement (`line 5`), i.e. that they are executed as atomic steps. (Such behavior seems natural for Ethernet-like networks which have a shared broadcast medium. One must ensure, however, that no overruns at input queues occur since this would violate the atomicity assumption.) Figure 6.2 gives a clock synchronization algorithm for clean-crash faults. We will now confirm that the algorithm reaches the same properties as the algorithm given in the previous section for $n \geq f + 1$.

The algorithm given in Figure 6.2 is a simplified version of the algorithm of Figure 6.1. Since we just consider clean-crash faults here, every message a process receives is received by all correct processes within ε by definition. Therefore the algorithm requires just one rule to update its clock value.

```
0:  VAR k : integer := 0;
1:
2:  /* Initialization */
3:  send (round 0) to all [once];
4:
5:  if received (round k) from at least n − f distinct processes
6:        → k := k + 1;
7:            send (round k) to all [once]; /* start next round */
8:  fi
```

Figure 6.2. Clock Synchronization Algorithm tolerating Clean-Crash Faults

The properties from Theorem 2 are extended from correct processes to all processes (before they stop); the properties are uniform.

Theorem 7 (Properties with Clean-Crash Faults). *In presence of $f < n$ clean-crash faults, the algorithm given in Figure 6.2 satisfies the following properties:*

- Uniform Progress. *If all correct processes set their clocks to k by time t, then every process sets its clock at least to $k+1$ by time $t+\tau^+$.*
- Uniform Unforgeability. *If no process sets its clock to k by time t, then no process sets its clock to $k+1$ by time $t+\tau_f$ or earlier.*
- Uniform Simultaneity. *If some process sets its clock to k at time t, then every process sets its clock at least to k by time $t+\varepsilon$.*

Proof. We show the properties separately.
Uniform Progress. By assumption at least $n-f$ correct processes send *(round k)* by time t. These messages are received by time $t+\tau^+$ by all processes (that are still alive) which then set their clocks to $k+1$ by `line 5` if they have not already done so.

Uniform Unforgeability. Assume by contradiction that a process sets its clock to $k+1$ by time $t+\tau_f$. Processes can update their clocks only by `line 5`. Thus the process has received at least $n-f$ *(round k)* messages. These messages must have been sent by t. Since processes send messages only for ticks less or equal their clock value we derive a contradiction.

Uniform Simultaneity. Let process p be the first process to set its clock to k. By Lemma 9 the first process has set its clock to k by time t using `line 5`. This happens based on $n-f$ *(round $k-1$)* messages by distinct processes, which must be received by all processes (that are still alive) by $t+\varepsilon$. They then set their clocks to $k-1$ by `line 5`.

If process p is not the first process which sets its clock to k, then at least one correct processes has done so before at time $t' < t$. By the same reasoning as above all correct processes must set their clocks to $k-1$ by time $t' + \varepsilon < t + \varepsilon$. □

Uniform simultaneity in the clean-crash fault case reconciles (Q) and (S) as defined in the Byzantine case. Thus (P), (U), (Q), and (S) apply. We may therefore state the following corollary which implies that the algorithm given in Figure 6.2 satisfies all derived properties (π) and (α) of the previous section as well.

Corollary 1. *In presence of $f < n$ clean-crash faults, the algorithm given in Figure 6.2 satisfies the properties of Theorem 2 according to the computational model in Section 3.5. In the case where $n = f + 1$, one must set $\tau_f = \tau^-$.*

Crash Faults

Crashes have the following impact: Processes may fail to send messages to all, i.e. they may send messages only to a subset of the processes. It is of practical importance that the properties from Theorem 2 must be uniform, i.e. that they must also apply to crash faulty processes until they actually crash.

The algorithm given in Figure 6.3 is very similar to the algorithm of Figure 6.1. We again require two rules for maintaining the clock value since due to crashes message may be received inconsistently by correct processes. A difference to the algorithm of Figure 6.1, however, is that every received message was sent by a (then) correct processes. Thus `line 5` has to wait for 1 message only instead of $f + 1$.

```
0:  VAR k : integer := 0;
1:
2:  /* Initialization */
3:  send (round 0) to all [once];
4:
5:  if received (round ℓ) from at least 1 process with ℓ ≥ k
6:     → k := ℓ; /* jump to new round */
7:         send (round k) to all [once];
8:  fi
9:
10: if received (round k) from at least n − f distinct processes
11:    → k := k + 1;
12:        send (round k) to all [once]; /* start next round */
13: fi
```

Figure 6.3. Clock Synchronization Algorithm tolerating Crashes

Lemma 9 (2nd if – Restricted Faults). *The first process that sets its clock to $C_{\max}(t) > 0$ by time t must do so by* `line 10`*.*

Proof. By contradiction. Assume that the first process p sets its clock to $k = C_{\max}(t)$ at instant t by `line 5`. At least one process must have sent a message for a tick $\ell \geq k$

to enable the rule at p. Since processes only send messages for ticks less or equal than their local clock value, at least one must already have a clock value $\ell \geq k$ at instant t. This contradicts p to be the first one that reaches clock value k. □

Theorem 8 (Properties with Crashes). *In the presence of $f < n/2$ crash faults, the algorithm given in Figure 6.3 satisfies the following properties:*

- Uniform Progress. *If all correct processes set their clocks to k by time t, then every process sets its clock at least to $k+1$ by time $t+\tau^+$.*
- Uniform Unforgeability. *If no process sets its clock to k by time t, then no process sets its clock to $k+1$ by time $t+\tau_f$ or earlier.*
- Uniform Quasi Simultaneity. *If some process sets its clock to k at time t, then every process sets its clock at least to $k-1$ by time $t+\varepsilon$.*
- Uniform Simultaneity. *If some process sets its clock to k at time t, then every process sets its clock at least to k by time $t+\tau^+ + \varepsilon$.*

Proof. We show the properties separately.
Uniform Progress. By assumption at least $n-f$ correct processes send *(round k)* by time t. These messages are received by time $t+\tau^+$ by all processes (that are still alive) which then set their clocks to $k+1$ by `line 10` if they have not already done so.

Uniform Unforgeability. Assume by contradiction that a process p sets its clock to $k+1$ by time $t+\tau_f$. Processes may set their clock by (1) `line 10` or (2) `line 5`.

Assume (1). Process p does so based on $n-f \geq f+1$ *(round k)* messages by distinct processes. By Lemma 1 at least one of these messages was sent by time t, which provides the required contradiction since processes send messages only for ticks less or equal their clock value.

Assume (2). Process p does so based on one *(round $k+1$)* message, i.e. at least one process has already clock value $k+1$ by time $t+\tau_f$. The first process which has set its clock to $k+1$ must have done so by `line 10` by Lemma 9. By applying (1) we again derive a contradiction.

Uniform Quasi Simultaneity. Let process p be the first process to set its clock to k. By Lemma 9 the first process has set its clock to k by time t using `line 10`. This happens based on $n-f$ *(round $k-1$)* messages by distinct processes. At least one of these must be sent by a correct process whose messages are received by all processes (that are still alive) by $t+\varepsilon$. They then set their clocks to $k-1$ by `line 5`.

If process p is not the first process which sets its clock to k, then at least one correct processes has done so before at time $t' < t$. By the same reasoning as above all correct processes must set their clocks to $k-1$ by time $t'+\varepsilon < t+\varepsilon$.

Uniform Simultaneity. According to uniform quasi simultaneity all processes set their clock to $k-1$ by $t+\varepsilon$. By uniform progress all processes set their clock to k by time $t+\tau^+ + \varepsilon$. □

As in the case of clean-crash faults the following lemma ascertains the derived properties (π) and (α) from Section 6.1.2 in the case of crash faulty processes.

Corollary 2. *In presence of $f < n/2$ crash faults, the algorithm given in Figure 6.3 satisfies the properties of Theorem 2.*

Remark The analysis in Section 7 reveals that the algorithm given in Figure 6.3 in fact also tolerates send omission faulty processes. These are processes that fail to send an arbitrary number of messages but are otherwise correct, i.e. they follow the algorithm correctly. Crash faulty processes can be regarded as a sub class of omission faulty ones as they fail to send all messages after they crash.

6.2 Failure Detection

In this section, we show how to extend the clock synchronization algorithm of Section 6.1 in order to obtain an implementation of the perfect failure detector $\mathcal{P}$. Note that it is possible to use even the clock synchronization algorithm for Byzantine faults in order to handle early timing faults (and hence improve coverage). Classic failure detectors, however, are defined for crash faults only. The following properties must be provided by every implementation of $\mathcal{P}$ [19]:

(SC) *Strong completeness*: Eventually, every process that crashes is permanently suspected by every correct process.

(SA) *Strong accuracy*: No process is suspected before it crashes.

The algorithm given in Figure 6.4 is a simple extension to the clock synchronization algorithm of Figure 6.1. The first addition is the vector $saw_max[\forall q]$ that stores for every process q the maximum clock tick k received via *(round k)* messages. It is written upon every message reception in `line 3`. Whenever a process updates its clock k (compare `line 5` and `line 10` in Figure 6.1), it checks $saw_max[\forall q]$ in `line 6` in order to find out which processes failed to send messages for tick $k - \Xi_{\mathcal{P}}$ at least. All those processes are entered into the vector $suspect[\forall q]$, which is the interface to upper layer programs that use the failure detector module, hence the list of suspects.

```
0:  VAR suspect[∀q] : boolean := false;
1:  VAR saw_max[∀q] : integer := 0;

2:  Execute Clock Synchronization from Section 6.1

3:  if received (round ℓ) from q
4:     → saw_max[q] := max(ℓ, saw_max[q]);
5:  fi

6:  whenever clock value k is updated do (after updating)
7:     → ∀q suspect[q] := (k − Ξ_P) > saw_max[q];
```

Figure 6.4. Failure Detector Implementation

We will now show that, if $\Xi_{\mathcal{P}}$ is chosen appropriately, the algorithm given in Figure 6.4 indeed implements the perfect failure detector.

Theorem 9. *Let $\Xi_{\mathcal{P}} \geq min\{\lceil 3\Omega+1 \rceil, \lceil 2\Omega+2 \rceil\}$. In a system with $n \geq 3f+1$ processes, the algorithm given in Figure 6.4 implements the perfect failure detector.*

Proof. We have to show that strong completeness (SC) and strong accuracy (SA) are satisfied.

For showing (SC), it is sufficient to notice that a process that crashes after it updates its clock to ℓ will be suspected by every correct process p when p reaches clock value $k \geq \ell + \Xi_{\mathcal{P}}$. Since progress of clock values is guaranteed by Lemma 5 every correct process will reach clock value k within bounded time (the exact time bound is derived below in Theorem 10).

To prove (SA), we have to show that $\Xi_{\mathcal{P}}$ is chosen sufficiently large such that every correct process which reaches a clock value k at time t has already received messages for ticks at least $k - \Xi_{\mathcal{P}}$ by every correct process. In the worst case setting, a correct process p sets its clock to k at instant $\sigma_p^{k-1} = \sigma_{first}^{k-1}$; hence $k = C_{max}(\sigma_p^{k-1})$. From Lemma 3, it follows that $C_{max}(\sigma_p^{k-1} - \tau^+) \geq k - \lceil \Omega \rceil$. Assuming a precision D_{max}, a bound for the smallest possible clock value of a correct process reads $C_{min}(\sigma_p^{k-1} - \tau^+) \geq C_{max}(\sigma_p^{k-1} - \tau^+) - D_{max} = k - \lceil \Omega \rceil - D_{max}$. Consequently, every correct process must have sent a message for tick $C_{min}(\sigma_p^{k-1} - \tau^+)$ by time $\sigma_p^{k-1} - \tau^+$, which arrives at p by time σ_p^{k-1}. Thus, choosing $\Xi_{\mathcal{P}} \geq \lceil \Omega \rceil + D_{max}$ is sufficient to ensure that p does not incorrectly suspect any correct process.

Since $\lceil x \rceil + \lceil y \rceil \geq \lceil x + y \rceil$ and $\lceil -x \rceil = -\lfloor x \rfloor$, it follows from setting $x = z + w$ and $y = -w$ that $\lceil z + w \rceil \geq \lceil z \rceil + \lfloor w \rfloor$. By setting $z = \Omega$ and $\lfloor w \rfloor = D_{max}$ we employ $\lceil z+w \rceil$ to choose $\Xi_{\mathcal{P}}$. Depending on Ω, precision D_s or D_q is smaller and can be used to calculate $\Xi_{\mathcal{P}}$ (by replacing D_{max}). Hence, using $D_s = \lfloor 2\Omega + 1 \rfloor$, we get $\Xi_{\mathcal{P}} \geq \lceil 3\Omega + 1 \rceil$. Performing the same calculation for $D_q = \lfloor \Omega + 2 \rfloor$, we get $\Xi_{\mathcal{P}} \geq \lceil 2\Omega + 2 \rceil$. □

To find the worst case detection time of our FD algorithm, we have to determine how long it may take from the time a process p crashes with clock value k until all correct processes reach a clock value of $k + \Xi_{\mathcal{P}}$ and hence suspect p. In the worst case setting, process p has the maximum clock value and crashes immediately after reaching it. All other processes must first catch up to the maximum value, and then make progress for $\Xi_{\mathcal{P}}$ ticks until correctly suspecting p.

Theorem 10 (Detection Time). *The algorithm of Figure 6.4 with the value $\Xi_{\mathcal{P}}$ chosen according to Theorem 9 implements the perfect failure detector. Its detection time is bounded by $(\Xi_{\mathcal{P}} + 2)\tau^+ - \tau^-$.*

Proof. Assume the worst case: A process p is crashing at time t_c where $k = C_p(t_c) = C_{max}(t_c)$. By (S) in Theorem 2 every correct process must reach clock value k by time $t' = t_c + \tau^+ + \varepsilon$. When a correct process reaches clock value $k + \Xi_{\mathcal{P}}$ it will suspect p. By Lemma 5 every correct process must reach that clock value by time $t = t' + \Xi_{\mathcal{P}}\tau^+ = t_c + \tau^+ + \varepsilon + \Xi_{\mathcal{P}}\,\tau^+ = t_c + (\Xi_{\mathcal{P}} + 2)\,\tau^+ - \tau^-$. □

The detection time in Theorem 10 is given in the context of the computational model of Section 3.3. In order to give the detection time in the system model of Section 3.2,

one has to transfer the times t_c and $t_c + (\Xi_{\mathcal{P}} + 2)\,\Omega - 1$ (see Equation 3.1) into real-time. The detection latency is hence given by $\Delta t = \beta^{-1}(t_c + (\Xi_{\mathcal{P}} + 2)\,\Omega - 1) - \beta^{-1}(t_c)$. When immersing the failure detector algorithm into a real system one has to consider $\bar{\tau}^+$ (compare Section 5.1) in order to derive the worst case detection time. Following the fast failure detector approach [51, 1] these detection times are much smaller than application level message delays. Using this approach it is hence possible to devise very efficient agreement algorithms.

Note that our failure detector implementation relies upon the totally time free clock synchronization algorithm from Section 6.1. The latter algorithm has no a priori knowledge of Θ such that violations of Θ do not do any harm to the correctness of the algorithm. Just the calculated bounds on (π) and (α) are violated. Consider the variant of the Θ-Model where Θ just holds from some unknown global stabilization time (GST) on [37, 19, 107]. Our clock synchronization algorithm will return to its predicted behavior within a short period of time.

Theorem 11 (Eventually Perfect Failure Detector). *When executed in a system where Θ (see system model, Section 3.2) holds from some unknown time GST on, the algorithm of Figure 6.4 with the value $\Xi_{\mathcal{P}}$ chosen according to Theorem 9 implements the eventually perfect failure detector $\diamond\mathcal{P}$.*

Proof. Let $k - 1 = C_{max}(GST)$. After time $t = \sigma^k_{first} + \varepsilon$ all correct processes have set their clocks based on messages by correct processes which were sent after GST. Thus (P), (U), (Q), and (S) apply for all rounds greater than k with values τ^+ and τ^- (as well as ε), satisfying the Θ assumption. Hence (π) and (α) are as predicted for these rounds. Thus Theorem 9 and Theorem 10 apply from time t on. □

Remark Note that we devised an alternative implementation [55] of $\diamond\mathcal{P}$ that works in networks with $n > f$ in the presence of up to f crash faults. It can also be used as a local FD [54] in sparse networks where just the neighbors have to be observed. A variant of this algorithm can be applied in order to get the leader oracle—the weakest FD which allows to solve asynchronous consensus [18]—both in sparsely and fully connected networks.

6.3 Synchrony of the Θ-Model

When taking a closer look at the implementation of $\mathcal{P}$ in Section 6.2 it turns out that is has stronger semantics than just (SC) and (SA): When a process detects a crash from process p it can also be sure that no more messages from p are in transit. This reveals that the Θ-Model allows a correct process p to determine an instant t when it is guaranteed that no more earlier messages from a given process will arrive at p after time t. A problem that can be reduced to the decision of whether a message from some process is still in transit is the strongly dependent decision problem [21]. In the following section we will give a solution for this problem.

6.3.1 The Strongly Dependent Decision Problem

Charron-Bost, Guerraoui and Schiper [21] showed that synchronous system models are strictly stronger that asynchronous systems enriched with (possibly perfect) failure detectors. The argument is that there are problems solvable in synchronous models that have no solution in asynchronous ones. To make the point clear, the strongly dependent decision (SDD) problem was introduced. There are two processes p and q. Process p has an input value v chosen from a set $\{0, 1\}$. Process q shall output a decision value satisfying the following requirements:

Integrity: Process q decides at most once.

Validity: If p has not initially crashed, the only possible decision value for q is p's initial value.

Termination: If q is correct, then q eventually decides.

The solution for synchronous systems is straight forward [21]. At round 0 process p sends its decision value to q. If p was not initially dead (stopped before round 0) q receives the message and decides during the computational phase of round 0. If p was dead, q detects this at round 0 and may decide any value (assuming lock-step synchrony where each round consists of send, receive, and computation phase).

In asynchronous systems there cannot be a solution to SDD: Even if a failure detector correctly detects a process p to be faulty, p may have sent a message when it was still alive. Since this message could travel arbitrarily long, a process q may decide based on the information provided by the failure detector and dismiss p's message. This would violate validity. If, on the other hand, q would ignore the suspicion of the failure detector while p was initially dead, q would wait vainly and termination would be violated.

We will now see that the Θ-Model is sufficiently strong to solve SDD, since the knowledge of Ω is sufficient to timeout a process. Informally our algorithm (given in Figure 6.5) works as follows. The initial value of the producer p is piggybacked to the *(round* 0*)* clock synchronization message. Processes p and q then execute the clock synchronization introduced in Section 6.1. When the consumer q reaches a round number $k \geq \Xi_{sdd}$ and it has not received a message from process p it can conclude that p was initially dead and that no *(round* 0*)* messages are in transit; q decides 0. If q receives p's *(round* 0*)* message, however, it may decide on p's value.

Since SDD is defined for clean-crash faults only, we use the clock synchronization algorithm for clean-crash faults given in Section 6.1.3 with $n = 2$ and $f = 1$, which satisfies (U), (Q), (P), and (S). Using these properties we now show that the algorithm given in Figure 6.5 in fact solves the SDD problem.

Theorem 12. *The algorithm given in Figure 6.5 with $\Xi_{sdd} \geq \lceil 2\Omega \rceil$ solves the SDD problem.*

For the producer process p

```
1:  send (round 0, v) to q [once];

2:  if received (round 0) from q then         /* if the message from the consumer is received  */
3:       → send (round 0, v) to q [once];     /* before line 1 was executed                    */
4:  fi

5:  Execute Clock Synchronization from Section 6.1.3
```

For the consumer process q

```
6:  send (round 0) to p [once];

7:  Execute Clock Synchronization from Section 6.1.3

8:  if received (round 0, v) from p then
9:       → return(v);
10: fi

11: if k > Ξ_sdd then
12:      → return(0);
13: fi
```

Figure 6.5. A Solution to the SDD Problem

Proof. We show the three requirements of SDD separately.

Integrity. Deciding is done using the return function, hence once a decision is made the function is left.

Validity. If p is not initially crashed it must send its *(round* $0, v$*)* message either spontaneously (upon booting) or as answer to q's *(round* 0*)* message. This message must be received before the time when process q's clock value $k > \Xi_{sdd}$, if Ξ_{sdd} is chosen sufficiently large. Let t be the time q sends *(round* 0*)*. It must receive *(round* $0, v$*)* from p by time $t + 2\tau^+$ (round trip). By Lemma 3 it may reach a clock value $k \leq \frac{2\tau^+}{\tau_f} = 2\Omega$ by then; hence validity is satisfied since $\Xi_{sdd} \geq \lceil 2\Omega \rceil$.

Termination. The latest possible time t_d when process q decides is when k reaches a value $k > \Xi_{sdd}$. By Lemma 5, $t_d \leq t + \tau^+(\Xi_{sdd} + 1)$. □

Remarks

- In the Θ-Model, the integer value Ξ_{sdd} is the software clock's equivalent of a $2\tau^+$ timeout in semi-synchronous systems, i.e. one round trip time.

- In our two processes setting, clock synchronization possibly degrades at q to sending messages to itself. Obviously this is no practical solution, but should just demonstrate the power of the model.

6.3.2 Fault-Tolerant Broadcast

We have seen that SDD is solvable in the Θ-Model, since it is possible to timeout certain events (the sending of a message). This leads us to the problem of atomic broadcast [50], where all receivers of messages must deliver messages in the same order. In order to do so *receive* and *deliver* must be distinguished: When a message drops in from the network we say it is received. It is then queued in the atomic broadcast algorithm until a decision is made whether the message should be accepted, in which case the message will be delivered to upper network layers. To implement atomic broadcast, we can hence use our timeout mechanism to determine locally that no messages are in transit that should be delivered before the messages which are queued for delivery.

In order to handle crash faults, we have to ensure that messages are either delivered by each correct process or by none (which would be the case in the presence of clean-crash faults by definition). Hence we have to ensure *Uniform Timed Reliable Broadcast* (UTRB) [12], which requires the following properties:

Integrity: For any message m, each process delivers m at most once, and only if m was actually broadcast by some process

Validity: If a correct process broadcasts a message m, then all correct processes eventually deliver m.

Timeliness: There is some time Δ_b such that if the broadcast of m is initiated at real-time t, no process delivers m after real-time $t + \Delta_b$.

Uniform Agreement: If a process delivers m, then all correct processes eventually deliver m.

Babaoğlu and Toueg [12] showed that the algorithm given in Figure 6.6 solves the UTRB problem with termination time $\Delta_b = (f+1)\tau^+$. We therefore leave the following theorem without proof.

```
/* to rb-send, the broadcaster executes: */
0:   send (message) to all processes;
1:   deliver (message);

/* to rb-deliver, process p ≠ broadcaster executes: */
2:   upon first receipt of (message) do
3:           send (message) to all processes;
4:           deliver (message);
```

Figure 6.6. A Simple UTRB Algorithm

Theorem 13 (Reliable Broadcast). *The algorithm given in Figure 6.6 achieves the properties required for uniform timed reliable broadcast.*

In addition to the properties of reliable broadcast, atomic broadcast requires that all processes deliver the messages in total order.

Total Order: If processes p and q both deliver messages m and m', then p delivers m before m' if and only if q delivers m before m'.

We will see that this can be achieved in the Θ-Model. In fact, the algorithm given in Figure 6.7 solves FIFO-Atomic-Broadcast (FIFO refers to the property that messages sent by process p are received in the same order as p has sent them). Note that the algorithm integrates the reliable broadcast algorithm from Figure 6.6.

```
/* Broadcaster s executes: */
0:  send (message, s, C_s, i) to all processes        /* C_s is the local time stamp */
1:  queue (message, s, C_s, i) for delivery           /* i is the sequence number   */

/* Process q ≠ broadcaster executes: */
2:  upon first receipt of (message, s, C_s, i)
3:        → send (message, s, C_s, i) to all processes
4:          queue (message, s, C_s, i) for delivery

/* Any process p executes: */
5:  whenever clock k is updated do (after updating)
6:        → deliver^a all queued messages with time stamp C_s ≤ C_p − Ξ_abc

7:  Execute Clock Synchronization from Section 6.1.3
```

[a] ordered by (1) increasing timestamps, (2) increasing process identifiers, and (3) increasing indexes i

Figure 6.7. A Simple Atomic Broadcast Algorithm in the Θ-Model

For the algorithm in Figure 6.7, we assume that all processes have synchronized clocks according to Section 6.1. Every process s that wishes to broadcast a message at time t, timestamps this message with its local clock value $C_s(t_{send})$ and assigns an index i to it, i.e. a local sequence number that makes every message unique. This is required since processes may broadcast multiple message during one round, i.e. the time stamp is not sufficient for total ordering. Then it initiates a reliable broadcast (`line 0-1`). Each correct process p which receives a messages forwards the message with its original time stamp, according to the reliable broadcast algorithm (`line 2-4`). Instead of delivering immediately, p waits for delivery until it reaches a local clock value $C_p = C_s(t_{send}) + \Xi_{abc}$ (`line 5-6`). If two processes initiate a broadcast with the same time stamp the process identifier has to be used in order to establish a consistent delivery order. Since all processes deliver the same message during the same local round, message are totally ordered.

Theorem 14. *The algorithm given in Figure 6.7 with* $\Xi_{abc} \geq \lfloor \Omega(f+1) \rfloor + D_q$ *solves the atomic broadcast problem.*

Proof. Assume process s reliably broadcasts a message m at time t_{send} with local time stamp $C_s(t_{send})$. By time $t_{send} + \tau^+(f+1)$ each correct process either queues m for delivery, or never does so otherwise (by Theorem 13). The maximum clock value of any correct process that enqueues m at some time t is $C_p(t) \leq C_s(t_{send}) + \lfloor \Omega(f+1) \rfloor + D_q$ by Lemma 3 and (π). Hence reliable broadcast has already terminated when the first correct process delivers m.

All processes deliver the same set of messages due to reliable broadcast. Moreover they deliver them ordered by time stamp, process identifier, and local sequence number, hence in total order. □

Remark Obviously, D_q may be replaced by D_{max} in Theorem 14. We used D_q in order to reuse the proof also in the case of booting—where just D_q holds—in Section 8.4.

6.3.3 Simulating Lock-Step

The simplest computational model for designing distributed algorithms is the model of lock-step synchrony: All processes execute round-based algorithms and make their receive, computation, and send actions simultaneously in lock-step. Physics, however, tells us that it is impossible to guarantee that events at distributed processes happen simultaneously. Still, it is possible to simulate lock-step behavior, i.e. create the illusion of lock-step synchrony to applications.

Such simulation is particularly convenient if we have to consider more severe fault classes such as Byzantine faults, where unreliable failure detectors are not sufficient to solve e.g. the consensus problem.

Lock step algorithms are executed in simultaneous rounds, which consist of three actions at each correct process. They are executed in the following order: $send_r \rightarrow receive_r \rightarrow compute_r$. For the ease of presentation our synchronizer algorithm calls the function *start* given in Figure 6.8, where first the messages of the current round are read, then the computational step is taken, and finally the messages for the next round are sent. Note carefully that all the steps are taken in the same order – just in the call $start(0)$ the operations read and compute are omitted in order to start with $send_0$, to remain consistent with literature [71].

```
For each participant
1: procedure start(r:integer)
2: begin
3:     if r > 0
4:         read round r − 1 messages;
5:         execute round r − 1 computational step;
6:     fi
7:     send round r messages;
8: end;
```

Figure 6.8. Lock-Step Framework using the Synchronizer in Figure 6.9

We will see that it is possible in the Θ-Model to simulate lock-step executions. That such a simulation exists is quite surprising at first sight – given that the system

model does not stipulate an upper bound on message transmission times. On the other hand, our solution to the SDD problem already classified the Θ-Model as an "almost synchronous one". We present an algorithm which serves as middle layer between clock synchronization and lock-step algorithms. Because this algorithm provides information on when to perform an action (and not on the faulty processes) we call this algorithm a synchronizer [10].

Our solution relies upon the idea that in the Θ-Model it is possible to timeout processes (in the SDD problem of Section 6.3.1 we had an algorithm for one round and one sender for clean-crash faults only). We do so in our algorithm given in Figure 6.9. As underlying service we use our clock synchronization algorithm which guarantees (P), (U), (Q), and (S) in the presence of Byzantine faults. The clock synchronization ticks are regarded as micro ticks, not visible to the upper layer application. The upper layer is provided with macro ticks r. We assume that upper layer application messages are piggybacked on the corresponding clock synchronization messages.

For each correct participant p

1: **VAR** r : integer := 0; // lock step rounds

2: send *(round* 0*)* to all [once];

3: Execute Clock Synchronization from Figure 6.1

4: upon first update of clock do
5: call start(0);

6: whenever the local clock C_p is updated do
7: **if** $(C_p - 1)/\Xi_{sync} > r + 1$
8: $\quad \rightarrow r := r + 1;$
9: $\qquad$ call start(r);
10: **fi**

Figure 6.9. Synchronizer for the Θ-Model

To implement a synchronizer we have to ensure that all messages from correct processes for the current round are received at any correct process when it starts its computation step. Simultaneity (S) provides us with an upper bound $\tau^+ + \varepsilon$ on the interval where all correct processes enter some round r. Since message delays are bounded by τ^+, we just have to ensure that the time between the first correct process entering round r and the first correct process entering round $r + 1$ is larger than $2\tau^+ + \varepsilon$. We do so in the following theorem.

Theorem 15. *The algorithm given in Figure 6.9 with $\Xi_{sync} \geq 3\Omega$ is a synchronizer.*

Proof. We have to show that correct lock-step executions are guaranteed, i.e. that lock-step round r messages are received before the $receive_r$ step is taken in the following invocation of $start$. Assume the first correct process executes $start(r)$ for $r \geq 0$ at time t_r. Further let ℓ be the micro tick correlated with round r. By simultaneity (S) all

correct processes execute $start(r)$ by time $t = t_r + \tau^+ + \varepsilon$ such that all corresponding messages are received by time $t = t_r + 2\tau^+ + \varepsilon$. A correct process executes $start(r+1)$ at some time t_{r+1} when it reaches micro tick $\ell + \Xi_{sync}$ by `line 7`. By Lemma 3, $t_{r+1} \geq t_r + \Xi_{sync}\tau_f \geq t_r + 3\Omega\tau_f = t_r + 3\tau^+ \geq t_r + 2\tau^+ + \varepsilon = t$, thus all round r messages sent by correct processes must be received when a correct process executes $start(r+1)$. □

Theorem 15 reveals that any problem that is solvable in synchronous systems has a solution in the Θ-Model as well. We believe that this result is interesting given that the system model does not stipulate upper bounds on message delays. However, termination times of solutions that use our synchronizer may not always be optimal. For example, reliable broadcast in synchronous systems requires $f + 1$ rounds. We would therefore have to ensure that $(f+1)(2\tau^+ + \varepsilon)$ time units have elapsed, while we have seen a solution in Section 6.3.2 where just $(f+1)\tau^+$ time units have to be timed out. In the following section we will introduce atomic commit algorithms which work in this more efficient way.

6.4 Non-Blocking Atomic Commitment

The atomic commitment problem originates in replicated distributed databases where—in order to keep the databases consistent—transactions have to be made permanent either at all sites or at none, atomically. Before executing an update all participants are asked whether it is acceptable to do the update. If just one replica says NO the common decision must be ABORT and no participant may execute the update. Atomic commitment is of course not restricted to databases. Especially in the field of critical systems one might think of applications where a NO of just one participant must be considered in order to prevent hazards. We therefore introduce a solution to this problem in this section.

It is obvious that atomic commitment cannot be defined properly in the presence of Byzantine faults. We hence just consider restricted failure modes here.

It has been shown [21] that synchronous systems dominate asynchronous systems enriched with failure detectors regarding problems that can be solved: Any algorithm that requires certainty whether a message from a process is still in transit or not cannot be solved in asynchronous systems augmented with failure detectors, since failure detectors can only detect if a process has crashed and not whether it managed to send messages before a crash (compare Section 6.3.1). This property is vital for non-blocking atomic commit protocols as well, since one has to enforce that if a process decides it has knowledge of all votes.

The problem of atomic commit, however, allows several kinds of definitions (i.e. non-triviality properties). In literature [46] several asynchronous solutions to variants of NBAC are described. As already observed in [21], systems that have a solution to the SDD problem allow more efficient NBAC algorithms. We confirm this observation by devising two atomic commit algorithms in the Θ-Model.

Equipped with a perfect failure detector, solving NBAC is straight forward. If there are no faults all votes must be exchanged and a decision must be made, based on the

votes. In the case of faults, the failure detector eventually suspects a crashed process. Each process waits until it has received votes by all or a process is suspected, and then another round is started to decide; see [46] for a detailed description. If there are any suspicions, failure detector based atomic commit protocols must in general ABORT. Therefore they are quite inefficient. In the following section we will provide a protocol that allows to COMMIT even if processes fail during the execution of the algorithm.

Note that our atomic commit protocols will be fully distributed in the sense that they do not have a central coordinator.

6.4.1 Clean-Crash Faults

We assume processes here that may just stop, i.e. can fail by clean-crashes only: Hence sending a message to all happens atomically. We will see in Section 6.4.2 that this assumption can be dropped by employing a reliable broadcast algorithm (recall Section 6.3.2) that simulates this behavior, at the cost of increased response times and increased number of processes. Note that clean crashes are natural faults in broadcast networks (Ethernet) provided that overruns in input queues can be avoided.

In this section we will provide an atomic commit protocol which commits if all correct processes that are initially up vote YES. It is hence possible to COMMIT in cases where protocols based on failure detectors would ABORT. Note that our protocol can easily be adapted to COMMIT only if all processes were initially up. In fact, since at the end of the protocol all correct processes have the same information on the system, any deterministic function will lead to agreement. Which behavior is actually required in the envisioned system is a design decision.

Before we introduce our protocol, let us first state the requirements of NBAC:

Integrity: Every participant decides at most once.

Validity: If a participant's decision is COMMIT, then all votes are YES.

Uniform Agreement: No two participants decide differently.

Termination: Every correct participant eventually decides.

These properties are the same in any atomic commit protocol. Protocols differ, however, in the non-triviality property. It defines in which cases—in practice that means how often—a protocol commits. Take for example protocols where processes are equipped with the eventually strong failure detector $\diamond\mathcal{S}$ [46]. A false suspicion could lead to ABORT although no failure occurred and all voted YES. Under our assumption (clean-crashes) we give a protocol that provides the following non-triviality property.

Non-Triviality-Clean-Crashes: If all participants that are not initially dead vote YES the outcome decision is COMMIT.

Here processes may stop during the execution of the protocol. As long as they voted YES it is possible to COMMIT.

In the algorithm given in Figure 6.10 all participants send their vote together with their *(round 0)* message to all. Different requests can be distinguished via the *req* field in the messages. We have chosen to start our algorithm with clock value 0 because it eases the presentation. If there is a clock synchronization algorithm that runs during the whole system operation, however, the request can be timestamped according the the local clock value as well. The respective clock values when decisions are taken have to be calculated based on the time stamps of the requests in this case. (compare atomic broadcast in Section 6.3.2).

In order to satisfy non-triviality we have to ensure that each process sends its vote to all. Processes that are not initially dead must send at least one message, i.e. the first message must include the proposed value.

```
For each participant
1:  send (round 0, req, myvote) to all [once];
2:  if received (round 0, req, vote) from any participant then
3:     → send (round 0, req, myvote) to all [once];
4:  fi

5:  Execute Clock Synchronization from Section 6.1.3

6:  if k > Ξ^s_ac then
7:     → if only votes YES have been received then
8:          → return(COMMIT)
9:        else
10:         → return(ABORT)
11:       fi
12: fi
```

Figure 6.10. NBAC Protocol Tolerating Clean-Crash Faults

Before we show that the algorithm given in Figure 6.10 in fact solves the NBAC problem we need a preliminary lemma.

Lemma 10. *A participant executing the algorithm given in Figure 6.10 with* $\Xi^s_{ac} = \lceil 2\Omega \rceil$ *has received all votes when it decides.*

Proof. Let the first process send its vote to all at time t. All other participants must receive all votes by time $t + 2\tau^+$ by `line 2`. By Lemma 3 the largest round number a correct process reaches by then is $\lfloor 2\tau^+/\tau_f \rfloor = \lfloor 2\Omega \rfloor$. Since a participant decides when it has reached round number $k > \Xi^s_{ac} = \lceil 2\Omega \rceil$ it must have received all *(round 0)* messages from all participants that are not initially down, i.e. by all participants that voted. □

Theorem 16. *The algorithm given in Figure 6.10 with* $\Xi^s_{ac} \geq \lceil 2\Omega \rceil$ *achieves NBAC.*

Proof. We show each of the requirements of NBAC separately.
Integrity: A participant decides when $k > \Xi^s_{ac}$. Since it uses the return function only once, integrity follows.

Validity: Lemma 10 states that if a participant decides, it has received all votes. By `line 7` it only decides COMMIT if all votes where YES.

Uniform Agreement: If a process decides, it has received all votes. Any two participants hence decide on the same set of messages and must therefore decide the same value.

Termination: Every correct process decides when it reaches round $k > \Xi^s_{ac}$. Let t be the time the first process sends its *(round* 0*)* message. By `line 2` all processes send *(round* 0*)* by time $t + \tau^+$, which is the assumption of (P). Therefore every correct process reaches round k by time $t + \tau^+ + \tau^+ \Xi^s_{ac} \geq t + \tau^+(2\Omega + 1)$ (see Lemma 5).

Non-Triviality: If all votes are YES, by `line 7` every outcome must be COMMIT. □

Remark From simultaneity (S) follows that all correct processes reach round $k > \Xi^s_{ac}$ within $\tau^+ + \varepsilon$, such that they commit within a short period of time. This is important in the context of real-time systems as well.

6.4.2 Crashes

It is well known that for systems where processes may crash during the execution of a step, reliable broadcast is required in order to keep the views of the remaining processes consistent (compare Section 6.3.2). To solve NBAC in this case, we employ an adaptation of the protocols from [83] which are based upon a semi-synchronous system model, where upper time bounds on transmission and computation steps are known, and the processes are equipped with local hardware clocks with known drift. We will see that our software clocks can be employed in the same manner, without knowing the bounds on transmission and computation steps.

Due to the more severe failure mode, it is not possible to satisfy the same non-triviality property as if only clean-crash failures had to be considered. This section's protocol achieves the following property:

Non-Triviality-Crashes: If all processes vote YES and no failure occurs, then the outcome decision is COMMIT.

As in the case of clean-crash failures we start with a preliminary lemma which guarantees equal sets of messages at all processes when the decision is made.

Lemma 11. *All participants executing the algorithm given in Figure 6.11 with* $\Xi^c_{ac} \geq \lceil \Omega(f + 2) \rceil$ *have delivered the same votes when they decide.*

Proof. If there are no faults and all vote YES by reliable broadcast all deliver these votes and decide in `line 9`. If there is one vote NO and one process delivers it, by reliable broadcast all deliver it and decide in `line 3`. We have to show that if one of these two cases happen, they happen at every process before `line 6` is enabled.

Let the first process enter round $\ell > \Xi^c_{ac}$ where it decides using `line 6` at time t'. By Lemma 3 the first process enters round 1 not after time $t \leq t' - \Xi^c_{ac}\tau_f$. By (Q) hence all processes send *(round* 0*)* by time $t + \varepsilon$ and reliably broadcast their vote. By Theorem 13 all the votes are delivered by time $t'' = t + \varepsilon + (f+1)\tau^+ < t + (f+2)\tau^+ = t + \Xi^c_{ac}\tau_f \leq t'$. Since $t'' < t'$ our lemma is true. □

For each participant
1: along with *(round* 0*)* reliably broadcast *(req, myvote)*;

2: Execute Clock Synchronization from Section 6.1.3

3: **if** a vote NO has been delivered
4: → return(ABORT)
5: **fi**

6: **if** $k > \Xi_{ac}^c$
7: → return(ABORT)
8: **fi**

9: **if** all votes are delivered and all are YES
10: → return(COMMIT)
11: **fi**

Figure 6.11. NBAC Protocol Tolerating Crashes

Theorem 17. *The algorithm in Figure 6.11 with* $\Xi_{ac}^c \geq \lceil \Omega(f+2) \rceil$ *achieves NBAC.*

Proof. We show each of the requirements of NBAC separately.
Integrity: A participant decides using a return function, hence only once.

Validity: By `line 9`.

Uniform Agreement: By Lemma 11.

Termination: Every correct process decides when it reaches round $k > \Xi_{ac}^c$. Let t be the time the last correct process sends its *(round* 0*)* message and hence its vote. Every correct process reaches round k by time $t + \tau^+ \Xi_{ac}^c$; see (P) in Lemma 5.

Non-Triviality: If no failure occurs every correct process delivers all votes before it reaches a clock value $k > \Xi_{ac}^c$ by Lemma 11. It commits by `line 9`. □

6.5 Related Work

6.5.1 Clock Synchronization

Clock synchronization in distributed systems is a very well-researched field, see [30, 93, 82, 92, 84, 73, 85] for an overview. In the field of self-stabilizing [28] clock synchronization much work has been conducted [33, 32, 78, 6] as well. There is even some work on Byzantine self stabilizing clock synchronization [25, 33, 26]. The algorithms introduced in this thesis are improvements regarding coverage of our previous work presented in [106, 109]. These algorithms are adaptations of the classic non-authenticated clock synchronization algorithm by Srikanth and Toueg [95].

Clock synchronization with booting in the Θ-Model has been introduced in [106]. In [109] we presented a clock synchronization algorithm that handles hybrid failure modes. Besides link failures (which we have not discussed yet) the whole range of possible process failure modes were investigated.

6.5.2 Unreliable Failure Detectors

Failure detectors are particularly attractive, since they encapsulate synchrony assumptions in a time free manner. Consensus algorithms using FDs are hence completely time free and thus share the *coverage maximization* property proper to purely asynchronous algorithms. Nevertheless, the question of coverage arises also when *implementing* a failure detector, which of course requires the underlying system to satisfy at least some synchrony assumptions. In fact, existing implementations of the perfect failure detector $\mathcal{P}$ rest upon knowing the bounds on the end-to-end transmission delays of messages and hence require a synchronous system.

Since it has been shown in [63] that perpetual FDs cannot be implemented in partially synchronous systems with unknown delay bounds [37, 19], perpetual accuracy properties like strong accuracy ("no processor is suspected before it crashes") are usually replaced by eventual ones ("there is a time after which correct processors are not suspected by any correct processor"). Many papers deal with the implementation of such eventual-type FDs [19, 61, 62, 22, 48, 40, 51, 15, 2]. Eventual properties are usually in conflict with the timeliness requirements of real-time systems, however. Nevertheless if an algorithm, designed to work with an eventual FD, uses a perpetual FD that provides bounded detection times, real-time properties can be achieved here as well.

In purely asynchronous systems, it is impossible to implement even eventual-type failure detectors. FDs with very weak specifications [44, 4] have been proposed as an alternative here. The heartbeat failure detectors of [4] do not output a list of suspects but rather a list of unbounded counters. Like $\diamond\mathcal{P}$, they permit to solve the important problem of quiescent reliable communication in the presence of crashes, but unlike $\diamond\mathcal{P}$, they can easily be implemented in purely asynchronous systems.

In view of the impossibility results of [29] and [63], it was taken for granted until recently that implementing perpetual failure detectors requires accurate knowledge of delay bounds and hence a synchronous system model. Still, the algorithms presented in [76, 75] reveal that perpetual FDs can be implemented in a time-free manner in systems with specific properties. For example, there is a time-free implementation of $\mathcal{P}$ in systems where it can be assumed a priori that every correct processor is connected to a set of $f+1$ processors via links that are not among their f slowest ones. The algorithm cannot verify whether the underlying system actually satisfies this assumption, however, and no design for implementing this property was given.

The failure detector presented in this thesis follows an idea that was originally developed by Le Lann and Schmid [67]. There exists subsequent work on this approach regarding coverage maximization [68] and system booting [108].

6.5.3 Synchronizers

In literature several approaches have been proposed to simulate synchronous systems, see also [9] for an overview on several simulation techniques.. Awerbuch [10] introduced the term *synchronizer* to describe algorithms that simulate synchronous networks in asynchronous ones. The algorithms he presented could be used in networks with arbitrary topologies, but only in the absence of faults.

Dwork, Lynch and Stockmeyer [37] devised several simulations for lock-step synchrony in the presence of partial synchrony for several types of process failures (including non-authenticated Byzantine). These simulations reach lock-step behavior eventually. In the global stabilization time model it is reached after the system has stabilized. In the case where the timing bounds are unknown the round durations are increased, and lock-step is reached when the rounds are sufficiently long.

Round-by-round fault detectors where introduced by Gafni [43]. Such fault detectors provide some process p—running a round based algorithm—with information about other processes which might have crashed. If all processes from whom no message for the current round has dropped in are suspected, the next round is entered.

6.5.4 Non-Blocking Atomic Commit

Consensus [9, 71] and non-blocking atomic commitment [83, 46, 38, 27, 12] are very similar agreement problems. Their relation is investigated in [47, 20] and Guerraoui shows in [47] that these problems are not comparable in asynchronous systems.

The uniform timed reliable broadcast algorithm we employed can be found in [12], where several other topics related to atomic commitment are discussed as well

Guerraoui and Schiper [46] presented decentralized non-blocking commit protocols. In contrast to the classical two phase commit and three phase commit these protocols do not have a central coordinator. The algorithm in [46] rely upon the eventually strong failure detector $\diamond\mathcal{S}$. In [83] several non-blocking atomic commitment algorithms are presented and investigated for several system models—from synchronous to asynchronous with failure detectors. Timed atomic commit was introduced in [27]. The algorithm that was presented there uses local hardware clocks in order to detect that timing constraints are violated.

Lower bounds regarding message complexity as well as timing complexity can be found in [38].

Chapter 7

Booting Clock Synchronization in the Perception-Based Fault Model

We showed in Section 6 that many important problems in distributed computing have time free solutions. Still, all the presented algorithms rest on the assumption that all correct processes are always up respectively up early enough such that they do not miss each other's messages. When considering real systems this assumption is too strong. In timed systems this assumption can easily be enforced by assuming that all processes boot within a known time interval. Setting a local timeout before starting to send messages suffices to create the illusion of always up processes. Since we consider time free solutions we cannot use a priori knowledge of real-time bounds, however. Consequently, we make no assumption on booting times.

It is well known [30] that, without authentication, no more than one third of the processes may be Byzantine to ensure the clock synchronization properties (π) and (α) in the presence of timing uncertainty. Since the threshold of one third cannot be guaranteed during booting, our goal is to handle system startup without further increasing the number of required processes.

In this section we present a clock synchronization algorithm which exhibits graceful degradation during the booting phase: Whereas (α) can only be guaranteed after booting has completed, (π) is ensured during whole system operation. Using this algorithm we show later in Section 8 how to transform the algorithms from Section 6 such that they provide (some of) their properties during the booting phase as well.

Moreover we will present our solution under a hybrid failure model, i.e. a failure model that incorporates multiple types of process failures as well as link failures. Until now we distinguished only very few types of process faults: Byzantine, stops and crashes. In recently emerging application domains (e.g. wireless networks) the probability of link failures dominates process failures. Mapping link failures to process failures is not a good choice here [88], since one quickly runs out of non-faulty processes in case of traditional failure models.

Unfortunately there exist many impossibility results which state that many problems in distributed computing have no solution if links are unreliable. These results, however, refer to unrestricted link failures only. Fair lossy links—if infinitely many

messages are sent over a link infinitely many are received—is an often alternative. Algorithms may just retransmit messages as often as required in order to generate the abstraction of perfect links. A more appropriate approach for real-time applications is tolerating link faults by space redundancy (adding processes) instead of time redundancy (retransmission of messages), however. This approach has recently been shown to work for the consensus problem [86, 91, 105, 16] and in clock synchronization [109].

Lets now turn to process failures: From a theoretical point of view the "all Byzantine" assumption (like in Section 6.1) is of course safe. When considering the probability of Byzantine faults compared to crash faults, however, it is obvious that making the Byzantine assumption for just tolerating crash failures is expensive ($n \geq 3f+1$ instead of $2f+1$). A hybrid failure model [109] allows to dimension the system according to the actual reliability requirements (and based on failure probabilities) since formulas for n are given that incorporate independent numbers of e.g. Byzantine, symmetric, omission faults.

In this section we present an overview of the hybrid perception based failure model of [87] and describe how to extend the clock synchronization algorithm from Section 6.1 in order to handle booting. The analysis follows the one in [109].

7.1 Perception Based Failure Model

This section contains an overview of the perception based failure model, which extends the model introduced in [88] by adding messages with history and proper handling of the startup phase. It consists of an execution model, a basic physical failure model, and a more abstract perception failure model. Both the physical and the perception failure model are *hybrid* ones [104, 11], i.e., distinguish several classes of failures. The advantage of a hybrid failure model is its improved resilience: Less severe failures can usually be handled with fewer processes than more severe ones.

We will omit a detailed description of the *physical failure model* [87]. It distinguishes several classes of time and value failures for both processes and links, and uses assertions like "at most ϕ_{av} processes may behave Byzantine in a single round". Due to the exploding number of possible combinations of time and value failures, it is not used for analyzing fault-tolerant algorithms, however. Its primary purpose is the analysis of the assumption coverage [64] in real systems, see [90] for an example.

The physical failure model can be mapped to a more abstract (and vastly simpler) *perception failure model*, which is similar in spirit to the round-by-round fault detector approach of [43]. It is a generalization of the synchronous model of [90, 91], and is solely based upon the local view (i.e. perception of failures) of every process in the system. The perception failure model is particularly well-suited for analyzing the fault-tolerance properties of distributed algorithms.

7.1.1 Execution Model

We consider a system of n distributed *processors* connected by a fully or partially connected point-to-point network. All links between processors are bidirectional, con-

sisting of two unidirectional channels that may be hit by failures independently. Links need not necessarily provide FIFO transmission. The system will execute a distributed round based algorithm made up of one or more concurrent *processes* at every processor. Any two processes at different processors can communicate bidirectionally with each other via the interconnecting links. Every processor is identified by a unique *processor id* $p \in \Pi = \{1, \dots, n\}$; every process is uniquely identified system-wide by the tuple (processor id, process name), where the *process name* N is chosen from a suitable name space. Since a process will usually communicate with processes of the same name, we will distinguish processes primarily by their processor ids and suppress process names when they are clear from the context. Our model, however, allows sender and receiver process to have different names as well.

Since we restrict our attention to round based algorithms, all processes execute a finite or infinite sequence of consecutive *rounds* $k = 0, 1, \dots$. In every round except the initial one $k = 0$, which is slightly different, a single process p may broadcast (= successively send) a single message—containing the current *round number* k and a *value* V_p^k depending upon its local computation—to all processes contained in p's current *receiver set* $\mathcal{R}_p^k \subseteq \{1, \dots, n\}$.[1] We assume that every (non-faulty) receiver q knows its current *sender set* $\mathcal{S}_q^k = \{p : q \in \mathcal{R}_p^k\}$ containing all the processes that should have sent a message to it, and that a process satisfying $p \in \mathcal{R}_p^k$ (and hence $p \in \mathcal{S}_p^k$) sends a message to itself as well. Note that this convention does not prohibit an efficient direct implementation of self-reception, provided that the resulting end-to-end transmission delay is taken into account properly.

Concurrently, for every round number ℓ, process p receives incoming round ℓ messages from its peer processes $\in \mathcal{S}_p^\ell$ and collects their values in a local array (subsequently called *perception vector*) $\mathcal{V}_p^\ell = \{V_p^{1,\ell}, \dots, V_p^{n,\ell}\}$. Note that $\mathcal{V}_p^\ell = \mathcal{V}_p^\ell(t)$ as well as its individual entries $V_p^{i,\ell} = V_p^{n,\ell}(t)$ are actually time dependent; we will usually suppress t, however, in order not to overload our notation. Storing a single value for each peer in the[2] perception vector is sufficient, since any receiver may get at most one round ℓ message from any non-faulty sender. The entry $V_p^{q,\ell} \in \mathcal{V}_p^\ell$ (subsequently called *perception*) is either $\emptyset$ if no round ℓ message from process q came in yet (or if $q \notin \mathcal{S}_p^\ell$), or it contains the received value from the first round ℓ message of process q. In case of multiple round ℓ messages from the same sender q, which must be faulty here, the receiver could also drop all messages and set $V_p^{q,\ell}$ to some obviously faulty value, instead of retaining the value from the first message.

Process p's current round k is eventually terminated at the *round switching time* σ_p^k, which is the real-time when process p switches from round k to the next round $k+1$. Note that round switching is event based—and part of the particular algorithm—in

[1] In this chapter , we use the following notation: "Anonymous" processes and round numbers are usually denoted by lowercase letters p, q and k, l, respectively. Process subscripts denote the process where a quantity like $V_q^{p,k}$ is locally available, process superscripts denote the remote source of a quantity. Calligraphic variables like $\mathcal{V}_p^r$ denote sets or vectors, bold variables like $\boldsymbol{\tau}$ denote intervals.

[2] Since we allowed multiple concurrent processes, there may of course be several different perception vectors on a processor. Any process—but at most one per processor—may send messages for a specific perception vector, but at most one process per processor may receive messages from it.

case of asynchronous (time free) systems but enforced externally in case of synchronous systems. At the round switching time, the value $V_p^{k+1} = F_p(\mathcal{V}_p^k(\sigma_p^k), \Sigma_p)$ to be broadcast by process p in the next round $k+1$ is computed as a function F_p of the round k perceptions available in $\mathcal{V}_p^k = \mathcal{V}_p^k(\sigma_p^k)$ and p's local state Σ_p at time σ_p^k.

The above execution model is generalized by introducing messages with arbitrary history size $0 \leq h \leq \infty$. History size $h > 0$ means that a round k message includes also the values broadcast in the h rounds $0 \leq k-h, \dots, k-1$ prior to the current round k (if any). If some round k' message from sender q arrives at process p while in round $k \leq k'$, all the still empty entries $V_p^{q,\ell} \in \mathcal{V}_p^\ell$ for $max\{k, k'-h\} \leq \ell \leq k'$ are filled with the appropriate values contained in the message. Note that such "late" perceptions are retained even when a proper round ℓ message drops in later (which could occur since we do not assume FIFO channels); unlike the arrival of two proper round k messages, this does not constitute an error.

Although round ℓ perceptions filled in upon reception of a round $k' > \ell$ message may be time faulty, in the sense that they should have arrived earlier (namely, in the process's proper round ℓ message), they are nevertheless useful for some algorithms: $h = \infty$ models full information mode protocols, whereas $h = 0$ corresponds to the standard situation. The case $h > 0$ allows to model a more flexible round switching, which allows to incorporate information from processes within $h+1$ rounds.

Remark In `line 6` resp. `line 13` of the algorithm of Figure 7.4, for example, we employ $h = 1$ for *echo*-messages to substitute round ℓ messages that were not received due to late booting and catch-up. In case of this particular algorithm, "late" perceptions need not be considered time faulty. Typically, however, a history size $h > 0$ must be accounted for in the failure model, see Section 7.1.3.

Formally, the essentials of the above execution pattern are captured by two specific events: $be_p^k = V_p^k(t_p^k)$ is process p's round k *broadcast event*, whereas $pe_q^{p,k} = V_q^{p,k}(t_q^{p,k})$ denotes process q's *perception event* of process p's broadcast event. Those events are related via their parameter values $V_q^{p,k} = V_p^k$ (which are equal if there is no failure) and their occurrence times $t_q^{p,k} = t_p^k + \delta_q^{p,k}$, where $\delta_q^{p,k}$ is the *end-to-end computational + transmission delay* between sender p and receiver q in round k. Note that the local round k computation $V_p^{k+1} = F_p(\mathcal{V}_p^k(\sigma_p^k), \Sigma_p)$ is also included in $\delta_q^{p,k}$ (or in $\delta_q^{p,k+1}$), which creates the abstraction of algorithms that consist of statements executed in zero time.

Figure 7.1 shows this relation. Note carefully that $\delta_q^{p,k}$ does not only depend upon the local computation and network load related to the particular round k message sent from p and q. As argued in [67, 68], scheduling issues make $\delta_q^{p,k}$ dependent upon any round k message that is sent and received by p and q.

According to our computational model in Section 3.3 again our model stipulates minimum and maximum values $\tau^- > 0$ and $\tau^+ < \infty$, not necessarily known to the algorithm, such that

$$\tau^- \leq \delta_q^{p,k} \leq \tau^+ \tag{7.1}$$

for any two well-behaved processes p, q connected by a non-faulty link. Note that

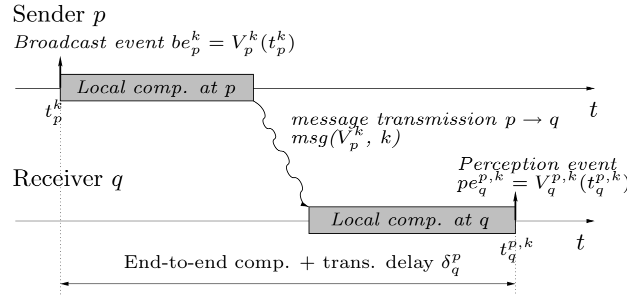


Figure 7.1. Relation between process p's round k broadcast event $be_p^k = V_p^k(t_p^k)$ and its corresponding perception event $pe_q^{p,k} = V_q^{p,k}(t_q^{p,k})$ at process q.

this relation must be valid for any round k, and for $p = q$ as well.[3] Introducing the interval $\boldsymbol{\tau} = [\tau^-, \tau^+]$, the above relation (7.1) can be written concisely as $\delta_q^{p,k} \in \boldsymbol{\tau}$. The resulting bound for $\delta_q^{p,k}$'s *delay uncertainty* resp. *delay ratio*, which will play a central role in our analysis, is given by $\varepsilon = \tau^+ - \tau^-$ resp. $\Theta = \tau^+/\tau^-$.

The above description of the execution model of a single process must still be extended to multiple processes. Three functions and one additional event are sufficient for this purpose:

- *create*(P) creates and initializes a process with name P on the caller's processor, including its perception vector and reception processing setup. It is usually called at boot time, for any process that will eventually run on a certain processor. If called at time t_q, it raises a parameterless init event ie_q^0 occurring at time $\iota_q^0 \geq t_q$ that signals completion of booting. It ensures that every correct process p's broadcast event that occurs at time $t_p^k \geq \iota_q^0 - \tau^-$ generates a perception event $pe_q^{p,k}$ at q at some time $t_q^{p,k} \geq \iota_q^0$, provided that the message is not hit by a link failure. Earlier broadcast event's are usually completely lost (at least must be assumed to be completely lost), however, in the sense that there is no corresponding perception event at q.

- *init*$(P, k, V_p^k, \mathcal{R}_p^k)$ raises the broadcast event be_p^k with receiver set $\mathcal{R}_p^k$ in process P at the caller's processor p. It can be called by any process running on the same processor as P and is also used internally by P to initiate be_p^k at the round switching time σ_p^{k-1} in round $k > 0$. Obviously, *init* must not be called more than once and only after ι_p^0.

- *wait*(P, k, X) blocks the caller process Q until process $P \neq Q$ on Q's processor has completed round k; a value X from P is optionally returned to the caller

[3]The typically very small self-reception delay could be considered as an (early) link timing failure and hence be masked by increasing f_ℓ^{ra} and f_ℓ^{sa} by 1.

upon unblocking. Note that this operation does not affect process P's execution and may require some memory if P terminates round k before *wait* is called.

In the pseudo code employed for the description of our algorithms, a more high-level way of expressing concurrency will usually be used:

```
cobegin
  { block_1 }
      ⋮
  { block_k }
coend
```

is a concise description of k concurrent processes, each executing one block_i.

7.1.2 Model of the Startup Phase

At the very beginning all processes are down. Every message that arrives at a process while it is down is lost, and no messages are sent by such a process. Note carefully that we do not even allow spurious messages from a down process or any of its links here. Correct processes boot at unknown times that cannot be bounded a priori.

During startup, a correct process goes through the following sequence of operating modes:

1. *down*: A process remains down when it has not been created yet or has not completed booting. Messages that drop in while a process is down are usually completely lost, i.e., do not generate a corresponding perception event.

2. *up*: A process q gets up when it has completed booting, at time ι_q^0. Recall that every correct process p's broadcast event that occurs at time $t_p^k \geq \iota_q^0 - \tau^-$ generates a perception event $pe_q^{p,k}$ at q at some time $t_q^{p,k} \geq \iota_q^0$, provided that the message is not hit by a link failure.

   (a) *passive*: A process that just got up performs an algorithm dependent initialization phase, where it is called passive. As the first action in passive mode, a process typically sends a special *join* message to all its peers. The first reception of a *join* message from some process p causes the receiver to send an algorithm-dependent reply message to p (point-to-point); subsequent *join* messages from the same sender are usually ignored.

   (b) *active*: A process that has completed its initialization phase is called active.

Remark In case of the algorithm of Figure 7.4, for example, a passive process broadcasts *join*—namely, *(round* 0*)*, in order to get the last *(round k)* message of every peer—and participates in the algorithm as in active mode. It need not satisfy the clock synchronization conditions (π) and (α) from Definition 3 while passive, however. The transition to active mode occurs when the process can be sure that it is within the synchronization precision D_{boot}.

The communication pattern for joining is obviously different from ordinary rounds. Whereas *join* is broadcast by a single joining process p, the replies are typically sent back by all peers via point-to-point messages. All the processes' *join* can be considered as a special booting round's messages, however, so both execution model and failure model apply to *join* as for any round k message. Response messages are different, however. Although compatibility with our execution model is maintained by assuming $\mathcal{R}_q^{resp} = \{p\}$ and $\mathcal{S}_p^{resp} = \Pi$, we need some additional assumptions with respect to our failure model.

Since the semantics of response messages are algorithm dependent, it does not make sense to define a generic model here. Basically, the joiner p must be provided with sufficient information to construct an internal state that is equivalent—in some algorithm dependent sense—to the state it would have had if it had not missed earlier messages due to late booting. This is a demanding requirement, however, since e.g. failures that occurred before p got up must be taken into account.

In case of the clock synchronization algorithm of Section 7.2, for instance, any process s that successfully received p's *join* responds by retransmitting[4] the last sent *(round k_s)*. Retransmissions must happen in a way, however, which lets the resulting perception matrices maintain compliance with our failure model. For example, if arbitrary link failures in process s's original round k_s broadcast already caused f_ℓ^{ra} receivers $q \neq p$ to deliver faulty round k_s perceptions V_q^{s,k_s}, it must not happen that s's response message to p's *join* suffers from an arbitrary link failure as well. Response messages may hence be viewed as messages belonging to the original round k broadcast here, which are deferred until the appropriate receiver is up. As in case of messages with history $h > 0$, perceptions recovered via response messages are usually time faulty, however (see Remark 4 on Definition 10).

7.1.3 Physical Failure Model

Our physical failure model basically[5] assumes that at most f_a, f_s, f_o and f_c processors may be arbitrary, symmetric, omission, and clean crash faulty during any single round. Clean crash faulty processes may be mute to all receiving processes (consistently) in some round, omission faulty processes may fail to send the correct message to an arbitrary subset of the receivers (inconsistently). Symmetric failures allow a process to disseminate an erroneous value consistently, whereas arbitrary faulty processes may even exhibit Byzantine behavior.

Since we will often establish *uniform* results (which also hold for processors with benign failures, namely, omission and clean crash faulty processes unless they have

[4] These retransmissions may cause the arrival of two round k messages from the same sender, which is usually considered an error in our execution model, recall Section 7.1.1. This is of course not true here, so we just assume that the first perception entered into some round k perception vector is retained—and not overwritten by some error value—when a second round k message drops in.

[5] In the fully-fledged physical failure model [87], it is differentiated between the number of physical failures $\phi_{av}, \phi_{sv}, \ldots$ and the number of perceptions failures $f_a, f_s, \ldots$ for generality. More specifically, formulas like $f_a = \phi_{av}$ for the perception failure numbers, given the physical failure numbers, are provided when mapping the the physical failure model to the perception failure model.

crashed), we will use the term *obedient* process to denote non-faulty, omission, and clean crash faulty processes that are still alive at the point of observation.

As far as link failures are concerned, *every* receiving process may experience at most f_ℓ^r link failures affecting different peers in the reception of *every* round, with at most $f_\ell^{ra} \leq f_\ell^r$ arbitrary time/value faulty ones (producing early or late message receptions and/or faulty values). Similarly, *every* sending process may generate at most f_ℓ^s link failures when sending a message to all its peers per round, with at most $f_\ell^{sa} \leq f_\ell^s$ arbitrary time/value faulty ones among those. Assigning different numbers of send and receive link failures is useful for modeling certain restricted process failures [90]; usually, however, $f_\ell^r = f_\ell^s = f_\ell$ and $f_\ell^{ra} = f_\ell^{sa} = f_\ell^a$. As long as processes do not exceed those limits, they need not be considered (process) faulty. Further details will be provided in the next section.

Remark Note that we stick to Schmid's [87] original definition of the fault model: Crash failures in the physical failure model are equivalent to clean-crash failures in Section 6.4.1. The (asymmetric) crash faults of Section 6.4.2 on the other hand must be modeled as omission faulty here.

7.1.4 Perception Failure Model

Consider the round k perception vector $\mathcal{V}_q^k(t)$—observed at some real-time t—of a well-behaved process q. Our execution model implies that $\mathcal{V}_q^k(t)$ is monotonic in time, in the sense that $|\mathcal{V}_q^k(t+\Delta t)| \geq |\mathcal{V}_q^k(t)|$ for any $\Delta t \geq 0$, since perceptions are only added. Moreover, since the value V_q^{k+1} to be broadcast in the next round $k+1$ is computed solely from $\mathcal{V}_q^k := \mathcal{V}_q^k(\sigma_q^k)$ and q's local state at the round switching time σ_q^k, it is obvious that, ultimately, only the failures in the perceptions present at the respective round switching times count. Timing failures are no longer visible here (but will probably affect σ_q^k, recall Section 7.1.1), since a message that did not drop in by σ_q^k at process q just results in $V_q^{p,k} = \emptyset$. Consequently, the resulting *perception failure model* is much simpler than the physical one and therefore more suitable for analyzing an algorithm's fault-tolerance properties.

Our formalization hence rests upon the $n \times n$ *perception matrix* $\mathcal{V}^k(t)$[6] of round k perceptions observed at the same arbitrary real-time t—typically some process's round switching time—at all processes:

$$\mathcal{V}(t) = \begin{pmatrix} \mathcal{V}_1(t) \\ \mathcal{V}_2(t) \\ \vdots \\ \mathcal{V}_n(t) \end{pmatrix} = \begin{pmatrix} V_1^1 & V_1^2 & \dots & V_1^n \\ V_2^1 & V_2^2 & \dots & V_2^n \\ \vdots & \vdots & \vdots & \vdots \\ V_n^1 & V_n^2 & \dots & V_n^n \end{pmatrix}_t \tag{7.2}$$

Note that $\mathcal{V}(t)$ is in fact a quite flexible basis for our failure model, since different "views" of the state of the distributed computation can be produced easily by choosing a suitable t.

We distinguish the following failure modes for single perceptions in $\mathcal{V}(t)$ in our perception failure model:

[6] We will subsequently suppress the round number k in quantities like $\mathcal{V}^k(t)$ for brevity.

Definition 7 (Perception Failures). *Process q's perception V_q^p of process p's broadcast value V_p can be classified according to the following mutually exclusive failure mode predicates:*

- correct(V_q^p)*:* $V_q^p = V_p$,
- omission(V_q^p)*:* $V_q^p = \emptyset$,
- value(V_q^p)*:* $V_q^p \neq \emptyset$ *and* $V_q^p \neq V_p$.

Next, we have to classify sender[7] process failures. This requires the important notion of *obedient* processes: An obedient process is an alive process that faithfully executes the particular algorithm. It gets its inputs and perform its computations exactly as a non-faulty process, but it might fail in specific ways to communicate its value to the outside world. We will subsequently use this term instead of non-faulty whenever a process acts as a receiver ("obedient receiver"), since this will allow us to reason about the behavior of (benign) faulty processes in case of uniform properties [50] as well. If $\mathcal{R}_p$ denotes some sender p's receiver set, let $\overline{\mathcal{R}}_p \subseteq \mathcal{R}_p$ denote the set of obedient processes among those.

Whereas the physical failure model differentiates timing failures according to $\delta_q^{p,k} \in \boldsymbol{\tau}$ vs. $\delta_q^{p,k} \notin \boldsymbol{\tau}$ and hence incorporates those quantities explicitly, it is solely the choice of t that is used in Definition 8 for this purpose: It only depends upon t whether a non-faulty perception V_q^p represents a perception event pe_q^p from a non-faulty or rather a timing faulty process p. Hence, neither $\delta_q^{p,k}$ nor $\boldsymbol{\tau}$ will show up in the definitions of the perception failure model below.

Definition 8 (Perception Process Failures). *Let p be a (faulty) sender process and g be some obedient receiver with* $\emptyset \neq V_g^p \in \mathcal{V}(t)$*, if there is any such g. In the absence of link failures, process failures of p can be classified according to the perceptions* $V_q^p \in \mathcal{V}(t+\varepsilon)$ *at all obedient receivers* $q \in \overline{\mathcal{R}}_p \subseteq \mathcal{R}_p$ *as follows:*

- *Non-faulty:* $\forall q \in \overline{\mathcal{R}}_p : \text{correct}(V_q^p)$,
- *Manifest:* $\forall q, r \in \overline{\mathcal{R}}_p : V_q^p = V_r^p \neq V_p$ *detectably,*
- *Crash:* $\forall q \in \overline{\mathcal{R}}_p : \text{omission}(V_q^p)$,
- *Omission:* $\forall q \in \overline{\mathcal{R}}_p : \text{correct}(V_q^p) \vee \text{omission}(V_q^p)$,
- *Symmetric:* $\forall q, r \in \overline{\mathcal{R}}_p : V_q^p = V_r^p$,
- *Arbitrary: no constraints.*

A faulty process producing at most omission failures is called benign faulty.

The following Definition 9 specifies the possible failures in perceptions caused by link failures. Note that it is formalized for non-faulty sender processes only, although link failures may of course also hit faulty senders in our final failure model.

[7] Receiver process failures will be considered below when introducing link failures.

Definition 9 (Perception Link Failures). *In the absence of sender process failures, a failure of the link from sender p to an obedient receiver q is classified according to its effect upon q's perception $V_q^p \in \mathcal{V}(t)$ as follows:*

- *Link non-faulty:* $V_q^p = V_p$,
- *Link omission:* $V_q^p = \emptyset$,
- *Link arbitrary: no constraint.*

The failure classes up to link omission failures are called benign.

To overcome the impossibility of consensus in presence of unrestricted link failures [45, 71], it turned out that send and receive link failures should be considered independently [91, 90]. The following link-failure-related parameters are hence incorporated in the final perception failure model of Definition 10 below:
($A1^s$) *Broadcast link failures*: For any single sender s, there are at most f_ℓ^s receiver processes q with a perception vector $\mathcal{V}_q$ that contains a faulty perception V_q^s from s caused by link failures, see Figure 7.2.

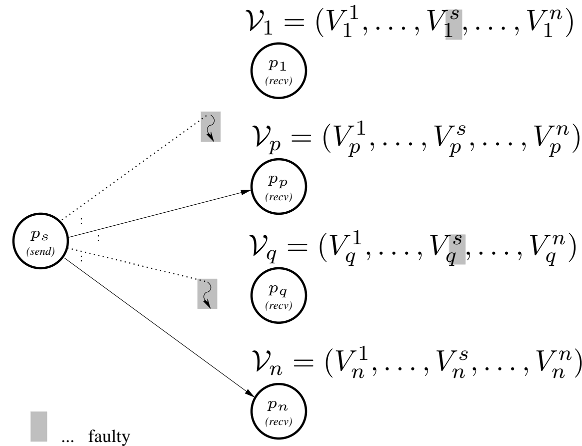


Figure 7.2. Example of a broadcast failure that affects the messages of two recipients.

($A1^r$) *Receive link failures*: In any single process q's perception vector $\mathcal{V}_q$, there are at most f_ℓ^r faulty perceptions V_q^p caused by link failures, see Figure 7.3.

Separating broadcast and receive link failures makes sense due to the fact that we consider the unidirectional channels, rather than the bidirectional links, as single fault containment regions: Broadcast link failures affect outbound channels, whereas receive link failures affect inbound channels. Still, broadcast and receive link failures are of course not independent of each other: If a message from process p to q is hit by a failure in p's message broadcast, it obviously contributes a failure in process q's message reception as well. Nevertheless, our failure model considers ($A1^s$) and ($A1^r$) as

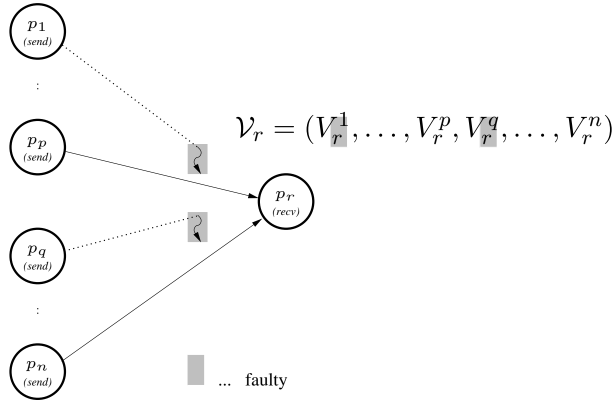


Figure 7.3. Example of a receive failure that involves the messages from two senders.

independent of each other and of process failures, for any process in the system and any broadcast/reception. Only the model parameters f_ℓ^s and f_ℓ^r cannot be independently chosen (without restricting the link failure patterns), since the system-wide number of broadcast and receive link failures must of course match. Hence, $f_\ell^s = f_\ell^r = f_\ell$ is the most natural choice, although other settings can also be considered [90].

Note carefully that we allow *every* process in the system to commit up to f_ℓ^s broadcast and up to f_ℓ^r receive link failures in every round, without considering the process as faulty in the usual sense. In addition, the particular links actually hit by a link failure may be different in different rounds. A process must be considered (omission) faulty, however, if it exceeds its budget f_ℓ^s of broadcast link failures in some round. Note that a process that experiences more than f_ℓ^r receive link failures in some round must in general be considered (arbitrary) faulty, since it might be unable to correctly follow the algorithm after such an event.

The following Definition 10 contains the complete perception based failure model, which just specifies the properties of any round's perception matrix $\mathcal{V}^R(t)$. Note carefully that this definition is valid for arbitrary times t, including those where the perceptions from some senders did not yet arrive (and are hence $\emptyset$).

Definition 10 (Asynchronous Perception Failure Model). *Let $\mathcal{V}^k(t)$ be the round k perception matrix of an asynchronous system of processes running on different processors that comply to our execution model. For any obedient receiver q, it is guaranteed that $V_q^{p,k} = \emptyset$ if $p \notin \mathcal{S}_q^k$ or if $V_q^{p,k}$ was not received by time t. Moreover:*

(P1) *There are at most f_a, f_s, f_o, f_c, and f_m columns in $\mathcal{V}^k(t)$ that may correspond to arbitrary, symmetric, omission, crash, and manifest faulty processes and may hence contain perceptions $V_q^{p,k}$ according to Definition 8.*

(A1^s) *In every single column p, at most f_ℓ^s perceptions $V_q^{p,k} \in \mathcal{V}^k(t)$ corresponding to obedient receivers $q \in \overline{\mathcal{R}}_p^k \subseteq \mathcal{R}_p^k$ may differ from the ones obtained in the absence*

of broadcast link failures. At most $f_\ell^{sa} \leq f_\ell^s$ of those perceptions may be link arbitrary faulty.

(A1^r) *In every single row q corresponding to an obedient receiver, at most f_ℓ^r of the perceptions $V_q^{p,k} \in \mathcal{V}^k(t)$ corresponding to senders $p \in \mathcal{S}_q^k$ may differ from the ones obtained in the absence of receive link failures. At most $f_\ell^{ra} \leq f_\ell^r$ of those may be link arbitrary faulty.*

(A2) *Process q can be sure about the origin p of $V_q^{p,k} \in \mathcal{V}^k(t)$.*

(A3) $\emptyset \neq V_q^{p,k} \in \mathcal{V}^k(t) \Rightarrow \emptyset \neq V_r^{p,k} \in \mathcal{V}^k(t+\varepsilon)$ *for every non-faulty sender p connected to obedient receivers q and r via non-faulty links.*

Remarks

1. The effects of process failures (P1) and link failures (A1^s), (A1^r) are considered orthogonal; it can hence happen that a link failure hits a perception originating from a faulty sender process. This is also true for manifest and clean crash failures, where link arbitrary failures could create non-empty perceptions at some receivers.
2. (P1) implies that our perception failure model is compatible with traditional process failure models. For example, f_a Byzantine faulty processes can generate inconsistently faulty perceptions only in the f_a columns they correspond to. Hence, all existing lower bounds and impossibility results, like $n \geq 3f_a + 1$ for consensus in presence of f_a arbitrary process failures [60], remain valid.
3. Analysis results under our perception failure model are automatically valid for partially connected networks as well, since the consequences of an incomplete communication graph can be viewed as (persistent) link omission failures. However, sparsely connected graphs and, in particular, partitioned ones cannot be modeled, cp. [90, Thm.3.2].
4. Our failure model also allows arbitrary resp. symmetric faulty processes to commit an early/late timing failure inconsistently resp. consistently (within ε). Neither omission faulty nor clean crash faulty processes may generate time-faulty perceptions in case of no history ($h = 0$). In case of messages with history or responses to *join*-messages, however, the latter cannot usually be guaranteed: Just take an omission faulty process s, for example, which omits sending a round k message to p but not to q. If this process sends a round $k+1$ message to p in the next round, it also generates a round k perception $V_p^{s,k}$ at p. Although $V_p^{s,k} = V_q^{s,k}$, the former is of course time faulty since it arrived too late w.r.t. the sender's proper round k's broadcast event. Hence, any type of benign process failure may result in a timing failure in case of $h > 0$ and must hence be accounted for in f_s or f_a. Note that the latter is not required for the algorithm studied in this paper, since late timing failures are no more severe than omissions for the algorithm of Figure 7.4.

5. Although message corruption is a fairly rare event if CRC checksums are used, one has to acknowledge the fact that the occurrence of arbitrary receive link failures is more likely than it meets the eye. After all, the messages sent in *every* round of the execution may be turned into a false round k message if the round number contained in the messages is corrupted.
6. Messages with history $h > 0$ can reduce the number of arbitrary link failures, since value failures may be detected by comparison and turned into omission failures here. The number of link omission failures is usually also decreased in case of $h > 0$. Like in the case of benign process failures, however, the number of late timing failures may increase and must be accounted for in f_ℓ^{ra} and f_ℓ^{sa}. Note that the latter is again not required for the algorithm studied in this paper, since late timing failures are no more severe than omissions for the algorithm of Figure 7.4.

The primary way of using our failure model in the analysis of agreement-type algorithms is the following: Given the perception vector $\mathcal{V}_q^R(\sigma_q^R)$ of some obedient receiver process q at its round switching time σ_q^R, it allows to determine how many perceptions will at least be present in any other obedient process r's perception vector $\mathcal{V}_r^R(\sigma_q^R + \varepsilon)$ shortly thereafter. The following Lemma 12 developed in [88] formalizes this fact.

Lemma 12 (Difference in Perceptions). *At any time t, the perception vector $\mathcal{V}_q(t)$ of any process at an obedient receiver q may contain at most $f_\ell^{ra} + f_a + f_s$ timing/value-faulty perceptions $V_q^p \neq \emptyset$. Moreover, at most $\Delta f = f_\ell^r + f_\ell^r + f_a + f_o$ perceptions V_r^p corresponding to $V_q^p \neq \emptyset$ may be missing in any other obedient receiver's $\mathcal{V}_r(t + \Delta t)$ for any $\Delta t \geq \varepsilon$.*

Proof. The first statement of our lemma is an obvious consequence of Definition 10. To prove the second one, we note that at most $f_\ell^{ra} + f_a + f_o$ perceptions may have been available (partly too early) at q without being available yet at r, additional $f_\ell^{ra\prime} \leq f_\ell^{ra}$ perceptions may be late at r, and $f_\ell^r - f_\ell^{ra\prime}$ ones could suffer from an omission at r. All symmetric faulty perceptions present in $\mathcal{V}_q(t)$ must also be present in $\mathcal{V}_r(t + \Delta t)$, however. Summing up all the differences, the expression for Δf given in Lemma 12 follows. □

7.2 The Algorithm

The clock synchronization algorithm considered here is a hybrid variant of the algorithm of Section 6.1. Note carefully that our algorithm is completely time free in that it does not incorporate τ^+ and τ^-, and not even ε or Θ.

The algorithm of Section 6.1 handles initialization if all obedient processes are up and listening to the network right from the beginning. Since we got rid of this assumption just executing the algorithm from Section 6.1 could lead to dead locks. We therefore add the booting section (`lines 2-5`) where processes just send *(round* 0*)* upon finishing booting, and answer such messages from others by retransmitting the last *(round k)* message they have sent. It is easy to see that round 1 can only be entered if a given

```
0:  VAR k : integer := 0; /* clock value */
1:  VAR mode : {passive, active} := passive;

cobegin
/* Booting Section */
2:  send (round 0) to all [once]; /* join message */
3:  if received (round 0) for the first time from p
4:     → re-send (round k) to p; /* current round */
5:  fi

/* Processes for (round ℓ) messages */
6:  if received (round ℓ) or (round ℓ + 1) from
    at least f_ℓ^ra + f_a + f_s + 1 distinct processes
7:     → if ℓ ≥ k
8:           → for j := k to ℓ − 1 send (round j) to all [once]; /* send skipped echos */
9:              k := ℓ; /* catch up to new clock value */
10:          fi
11:          send (round k) to all; [once]
12: fi
13: if received (round ℓ) or (round ℓ + 1) from
    at least n − f_ℓ^s − f_ℓ^r − f_a − f_s − f_o − f_c distinct processes
14:    → if mode = passive → mode := active; fi
15:       if ℓ ≥ k
16:          → for j := k to ℓ − 1 send (round j) to all [once]; /* send skipped echos */
17:             k := ℓ + 1;
18:       fi
19:       send (round k) to all; [once]
20: fi
coend
```

Figure 7.4. Clock Synchronization Algorithm for the Hybrid Perception Based Failure Model with Startup

number of correct processes are eventually up. We will later see that these processes are sufficiently many to guarantee (Q) quasi simultaneity and (U) unforgeability (compare Theorem 2), such that they always remain within precision (π) to each other. The progress property (P) however, cannot be guaranteed until all correct processes become up. Consequently, three consecutive modes of system operation must be distinguished to properly handle system startup:

- *Early mode*, where the first few correct processes have completed booting and started exchanging messages. It terminates when the first obedient process advances its clock to 1.
- *Degraded mode*, where enough correct processes are up such that some clocks may advance when "assisted" by faulty processes and links.
- *Normal mode*, where sufficiently many correct processes are up and synchronized to guarantee progress for all clocks.

Note carefully that it is impossible for any process in the system to delimit the exact borders between those modes from local information.

In order to add system startup handling to the original clock synchronization algorithm, *join* messages and an additional **if**-clauses are required. First of all, a newly booted process must tell all others that it is up now and must learn their current clock values. This is accomplished by means of *join* messages, as introduced in Section 7.1.2: Every process p sends *join* = *(round* 0*)* as the very first message after having completed booting. Every process q that receives this message replies by retransmitting its previously sent *(round k)* message (`lines 2-5`). This ensures that p will eventually get sufficiently many messages—which may have been lost while it was down—to trigger the catch-up rule described below.

In order to ensure that a late starter can catch-up it must receive $f_\ell^{ra} + f_a + f_s + 1$ messages for a single round from distinct processes (`line 6`). Due to (Q) we can only guarantee that correct processes remain within two rounds. This is why we introduced messages with history size $h = 1$.

Moreover, due to failures or a large group of simultaneous late joiners, the first round an initializing process might reach by the catch-up rule could be arbitrarily small. This would violate the precision requirement, however. We therefore assume that only active, but not passive processes, must satisfy (π). A passive process switches to active mode when it has sufficient evidence that its local round number is within precision D_{boot} of the clocks of the other correct active processes in the system. This is accomplished by the second **if**-clause (`line 13`), which triggers when sufficiently many *(round ℓ)* messages have been received. We will show in Section 7.4 that any such ℓ can be guaranteed to be sufficiently close to the maximum correct clock value in the system.

In the following sections, we will prove that our algorithm satisfies the clock synchronization properties (π) and (α) – (π) during whole system operation and (α) when sufficiently many correct processes are up. In Section 7.3, we will see how its execution matches the execution model of Section 7.1.1. The analysis starts with Section 7.4, where we explore how our algorithm behaves in early and degraded mode, i.e. when not sufficiently many processes are up to guarantee progress of the clocks. Section 7.5 deals with the transition between degraded and normal mode, which takes place when late processes join the system.

7.3 Mapping to the Perception based Execution Model

The algorithm is given in an event based notion that consists of the three outermost **if**-statements, numbered from the zero-th **if** (`line 2`) up to the second **if** (`line 13`). The variable k provides the clock value $C_p(t)$ of the processor p that executes the algorithm. In order to be able to analyze our algorithm under the perception based failure model of Definition 10, we must show that its execution complies to the execution model of Section 7.1.1. Every execution of our algorithm can in fact be described by the following process running on every processor p:

Infinitely many single-round processes L, $L \geq 0$, where process L processes *(round ℓ)*

messages for $\ell = L$ in the perception vector $\mathcal{V}_p^\ell$. All those processes must be active concurrently right from the start and can terminate after execution of `line 13`. Note that they can be viewed as a way of concurrently executing all rounds of a single multiple-round process.

In order formally model the fact that computations are actually triggered upon arrival of *(round x)* or *(round $x+1$)* in our algorithm, we use messages with history size $h = 1$. For example, when an *(round $k+1$)* message from process p arrives at q in case of $h = 1$, both the perception $V_q^{p,k+1}$ and, provided that still $V_q^{p,k} = \emptyset$, the perception $V_q^{p,k}$ is filled. Hence, the above processes can act upon a single perception vector $\mathcal{V}_q^k$ as required by our execution model, yet process messages *(round k)* and *(round $k+1$)*.

As in Definition 2 it is possible also for the perception based failure model to define an effective uncertainty ratio Ω. In order to do so we give the following definition and lemma.

Definition 11 (Short Transmission Times). *Let q be any obedient processes. Considering a fixed obedient process q, τ_q^- is the $n - f_\ell^s - f_\ell^{ra} - f_\ell^r - 2f_a - 2f_s - f_o - f_c$ smallest δ_{pq} for all messages sent by obedient processes p which are used to enable an event at q at real-time t. τ_f is the smallest τ_q^- for any obedient process q.*

The term $n - f_\ell^s - f_\ell^{ra} - f_\ell^r - 2f_a - 2f_s - f_o - f_c$ accounts for the minimal number of messages by obedient processes which is required by any obedient process—executing the algorithm given in Figure 7.4—to advance its clock by `line 13`.

Lemma 13 (Incoming Messages). *If a correct process receives messages by at least $n - f_\ell^s - f_\ell^{ra} - f_\ell^r - 2f_a - 2f_s - f_o - f_c$ distinct obedient processes by time t, then at least one message from these processes was sent by time $t - \tau_f$.*

Proof. Follows from Definition 1. □

Definition 12 (Hybrid Effective Uncertainty Ratio). *The smallest value of the uncertainty ratio which may be considered in the late binding process is $\Omega = \tau^+/\tau_f$.*

7.4 From Early to Degraded Mode

Before considering the full system startup scenario, we will study the algorithm of Figure 7.4 in a system of $n \geq f_\ell^s + 2f_\ell^{ra} + 2f_\ell^r + 3f_a + 2f_s + 2f_o + f_c + 1$ processes,[8] where a fixed but arbitrary number $0 \leq n_{up}(K) \leq n$ of processes have completed booting when the messages from the first obedient process that started some round K may arrive. Note that all those $n_{up}(K)$ processes can receive all messages for ticks $k \geq K$ from each other (except faulty ones, of course). The special case $n_{up}(0) = n$ describes the

[8]Incorporating f_ℓ^s in our algorithm is only required to guarantee liveness in the transition from degraded to normal mode (Theorem 22 and Theorem 24); it accounts for send link failures in *join*-messages of late starters. Consequently, most of our results—in particular, the ones of this section—hold when setting $f_\ell^s = 0$ in Figure 7.4 and in our formulas (despite of $f_\ell^r > 0$) as well.

situation where all processes must be up and running right from the very beginning, i.e., where system startup is ignored [70, 95, 86]. The case $n_{up}(1) < n$ will be used to describe the behavior of our algorithm during (the beginning of) degraded mode.

We start with some frequently used definitions and lemmas, which apply to any operation mode. Note that a passive process p's local clock $C_p(t)$ refers to its current round number k (see Figure 7.4) at time t in our analysis; terms like *p's clock value* and *"p sets its clock"* are hence meaningful for both active and passive processes. Still, the clock synchronization properties need only hold for active processes. Of particular importance is the maximum clock value of all obedient processes in the system, which will be used as the "reference time" in our precision analysis.

Definition 13 (Maximum Clock Value). *Consider some real-time t.*

- *$C_{\max}(t)$ is the maximum of the local clocks of all obedient processes that are up at time t.*
- *$\sigma_{first}^{k-1} = \sigma_p^{k-1} \leq t$ denotes the real-time when the first obedient process p sets its local clock to $k = C_{\max}(t)$; σ_{first}^{-1} is the time when the first obedient process has completed booting. Note that p's clock value, and hence $C_{\max}$, is defined to be k at time σ_{first}^{k-1}, and $k-1$ at $\sigma_{first}^{k-1} - \epsilon$ for any infinitesimally small $\epsilon > 0$.*

The maximum set of obedient processes that did not miss any tick k message from each other (except the ones missing due to process or link failures, of course) is denoted by $\mathcal{P}_{up}(k)$, with $n_{up}(k) = |\mathcal{P}_{up}(k)|$.

The following Lemma 14 reveals a few straightforward properties about messages sent by correct[9] processes.

Lemma 14 (Message Generation). *Consider any correct process p executing the algorithm of Figure 7.4. If p reached clock value $k > 0$ by time t, it must have sent (round k') for all $0 \leq k' \leq k-1$ by time t.*

Proof. Consider the sequence of values $\{c_m\}_{m\geq 0}$, assigned to p's clock variable at real-times $\{t_m\}_{m\geq 0}$. Using induction on m, we will show that p must have emitted *(round ℓ')* for all $0 \leq \ell' \leq c_m$ by time t_m. Since it is obvious from Figure 7.4 that c_m can only be monotonically increasing, our lemma then follows immediately.

For the basis $m = 0$, obviously $c_0 = 0$ and the claim is void since no *(round)* message needs to have been sent by t_0. Assuming now that the claim holds for $m-1 \geq 0$, i.e. all *(round ℓ')* for $0 \leq \ell' \leq c_{m-1}$ have been sent by time t_{m-1} when p's clock is set to c_{m-1}, we have to show that the claim holds for m as well: Two cases must be distinguished here, according to whether setting p's clock to c_m occurs via `line 6` or `line 13`.

If p sets its clock via `line 13` in in Figure 7.4, we must obviously have $c_m = c_{m-1}+1$. Since triggering of `line 13` implies triggering of `line 6`, *(round $c_m - 1$)* $=$ *(round c_{m-1})* must have been sent by time t_m. Combining this with the induction hypothesis, the claim follows immediately.

[9] Transmission of messages of obedient process q are guaranteed to be successful only if q is correct.

If p sets its clock to c_m via `line 6`, it sends *(echo, l'')* for $c_{m-1} \leq l'' \leq c_m$ by `line 8`; note that *(echo, c_{m-1})* is of course only emitted here if this has not happened earlier. Combining this with the induction hypothesis, our claim follows also in this case. □

The following lemma shows that progress of $C_{\max}(t)$ is only possible via `line 13`.

Lemma 15 (2nd if). *The first obedient process that sets its clock to tick $k = C_{\max}(t) > 0$ by time t must do so by* `line 13`*.*

Proof. By contradiction. Assume that the first obedient process p sets its clock to $k = C_{\max}(t)$ at instant t by `line 6`. At least one obedient process must have sent a message for a tick $\ell \geq k$ to enable this rule at p. Since obedient processes only send messages for ticks less or equal their local clock value, at least one must already have had a clock value $\ell \geq k$ at instant t. This contradicts p to be the first one that reaches clock value k. □

The following Lemma 16 shows that processes which completed booting early enough can be guaranteed to be in $\mathcal{P}_{up}(k)$, since they must have got all tick k messages from other obedient processes (except messages hit by failures). Bear in mind, however, that early booting is not the only only way of ensuring inclusion in $\mathcal{P}_{up}(k)$. A late starting process p may also achieve $p \in \mathcal{P}_{up}(k)$ via the *join*-protocol even though it completes booting after time $\sigma^{k-1}_{first} + \tau^-$.

Lemma 16 (Booting Times). *Any obedient process p that is up by time $\sigma^{k-1}_{first} + \tau^-$ satisfies $p \in \mathcal{P}_{up}(k)$.*

Proof. Since the first obedient process that emits a tick k message does so not earlier than time σ^{k-1}_{first}, according to our algorithm in Figure 7.4 and Lemma 15, no obedient process p can receive such a message before time $\sigma^{k-1}_{first} + \tau^-$. Since p is up by that time, it will hence receive all tick $\ell \geq k$ messages from correct processes. □

The following Theorem 18 is the first major result of this section. It shows that the clocks of obedient processes obey two properties, namely, uniform unforgeability (U), and uniform quasi simultaneity (Q) – in concordance with Section 6.1. These properties are weak in the sense that they suffice to ensure (π) but are to weak to guarantee (α). This is due to the small number of processes that participate during booting and the impossibility results [30] on the minimal number of processes which are required to guarantee (π) and (α).

Theorem 18 (Weak Clock Synchronization Properties). *For $n \geq f^s_\ell + 2f^{ra}_\ell + 2f^r_\ell + 3f_a + 2f_s + 2f_o + f_c + 1$, the algorithm from Figure 7.4 achieves*

(U) Uniform Unforgeability. *If no obedient process sets its clock to k by time t, then no obedient process sets its clock to $k+1$ by time $t + \tau_f$ or earlier.*

(Q) Uniform Quasi Simultaneity. *For any $k \geq 1$, if an obedient process sets its clock to k at time t, then, by time $t + \varepsilon$, every obedient process $\in \mathcal{P}_{up}(k-1)$ sets its clock at least to $k-1$ and, if correct, sends (round $k-1$).*

Proof. Uniform Unforgeability. Setting the clock value can be done by (1) the `line 13` or (2) `line 6`. The proofs are by contradiction.

Assume (1) that there is an obedient process p that sets its clock to $k+1$ before instant $t+\tau_f$ using `line 13`. This implies $|\mathcal{V}_p^k(t+\tau_f)| \geq n - f_\ell^s - f_\ell^{ra} - f_\ell^r - 2f_a - 2f_s - f_o - f_c \geq 2f_\ell^{ra} + f_\ell^r + 2f_a + f_o + f_s + 1$. Since, according to Lemma 12, only at most $f_\ell^{ra} + f_a + f_s$ of the corresponding *(round k)* perceptions may be due to messages produced by arbitrary receive link failures and time/value-faulty processes, at least one obedient process q must have sent *(round $\overline{k}$)* for $\overline{k} \in \{k, k+1\}$ before time $t+\tau_f$ (according to Lemma 13), which contradicts the assumption of unforgeability.

Assume (2) that there is an obedient process p that sets its clock to $k+1$ before instant $t+\tau_f$ using `line 6`. Process p does so because $|\mathcal{V}_p^l(t+\tau_f)| \geq f_\ell^{ra} + f_a + f_s + 1$ for some $l \geq k+1$. That is, at least one *(round $\overline{l}$)* message, $\overline{l} \in \{l, l+1\}$, must have been sent by an obedient process q before t. $\overline{l} > k$ and messages for tick $\overline{l}$ are sent by an obedient process only after tick k messages. But by assumption of (U), no tick k message was sent before t, which again provides the required contradiction.

Uniform Quasi Simultaneity. Assume first that $k = C_{\max}(t)$. If the obedient process p that advanced its clock to k by assumption used `line 13`, then $|\mathcal{V}_p^{(k-1)}(t)| \geq 2f_\ell^{ra} + f_\ell^r + 2f_a + f_o + f_s + 1$. Hence, the perception vector for tick $k-1$ at any obedient process $q \in \mathcal{P}_{up}(k-1)$ at time $t+\varepsilon$ must satisfy $|\mathcal{V}_q^{(k-1)}(t+\varepsilon)| \geq f_\ell^{ra} + f_a + f_s + 1$ according to Lemma 12.[10] It follows that every obedient process $\in \mathcal{P}_{up}(k-1)$ achieves sufficient evidence in `line 6` to catch-up to round $k-1$ (if it has not already done so). This fact (or Lemma 14 if p has already reached an even larger clock value $\geq k$ by $t+\varepsilon$) guarantees that *(round $k-1$)* is sent by time $t+\varepsilon$.

If, on the other hand, the obedient process p advanced its clock to k by `line 6`, there must be a process p' that set its clock to k by `line 13` earlier according to Lemma 15. Using p' in the above argument establishes (Q) in this case as well.

If $k < C_{\max}(t)$, then at least one obedient process has already set its clock to $l > k$ at some time $t' \leq t$ using `line 13`. We have shown in the previous paragraph that all obedient processes $\in \mathcal{P}_{up}(k-1)$ must set their clocks to $l-1$ (and hence at least to $k-1$) by time $t'+\varepsilon$ (and hence by time $t+\varepsilon$) as asserted. □

Since (U) and (Q) are satisfied, Lemma 3 and Lemma 4—derived in Section 6.1—hold as well. These lemmas will be used in the following analysis.

The following two technical Lemmas 17 and 18 are the major tools in our precision analysis. Both lemmas—and hence all the results built upon those—actually hold for every algorithm that satisfies uniform unforgeability (U) and uniform quasi simultaneity (Q). Lemma 18 bounds the maximum real-time difference between any obedient process's clock and $C_{\max}$ (i.e., the fastest obedient process) at about the same clock time. Note carefully that it actually holds for any $k' \geq k$ and hence for any $t \geq \sigma_{first}^k$, since $p \in \mathcal{P}_{up}(k)$ implies $p \in \mathcal{P}_{up}(k')$.

---

[10]If n is increased, $|\mathcal{V}_q^{(k-1)'}(t+\varepsilon)|$ increases accordingly. This fact will be used for ensuring quasi simultaneity (Q) despite of some missing *(round $k-1$)* due to send link failures: By adding another f_ℓ^s to n, the at most f_ℓ^s lost *join*-messages of a late starter p, which may lead to the lack of up to f_ℓ^s responses *(round $k-1$)* at p, can be masked transparently, see the proof of Theorem 22.

Lemma 17 (Real-Time Bounds). *For any obedient process $p \in \mathcal{P}_{up}(k)$ where $k+1 = C_{\max}(t)$ such that $\sigma^k_{first} \leq t < \sigma^{k+1}_{first}$, the following holds:*

$$\begin{aligned} \sigma_p^{k-1} &\leq \sigma^k_{first} + \varepsilon && (7.3)\\ C_p(t+\varepsilon) &\geq C_p(\sigma_p^{k-1}) = C_{\max}(\sigma^k_{first}) - 1 = C_{\max}(t) - 1 && (7.4) \end{aligned}$$

Moreover, if p is correct, it is guaranteed to send (round k) by time $t+\varepsilon$.

Proof. According to (Q) in Theorem 18, all obedient processes $\in \mathcal{P}_{up}(k)$, and hence also p, must set their clocks to at least k by time $\sigma^k_{first} + \varepsilon$. (Q) hence implies (7.3) and the claimed sending of *(round k)*.

If (7.3) holds, it follows from $\sigma^k_{first} \leq t < \sigma^{k+1}_{first}$ that $C_{\max}(t) = k+1$ and $\sigma_p^{k-1} \leq \sigma^k_{first} + \varepsilon \leq t+\varepsilon$. Consequently, $C_p(t+\varepsilon) \geq C_p(\sigma_p^{k-1}) = k$ and hence $C_{\max}(t) - C_p(t+\varepsilon) \leq 1$, which confirms (7.4). □

The following lemma provides a bound for the maximum clock time difference of any obedient process and $C_{\max}$ at the same real-time. Note that the precision D_q implied by Lemma 18 actually holds for any $k' \geq k$ and hence for any $t \geq \sigma_p^{k-2}$, since $p \in \mathcal{P}_{up}(k)$ implies $p \in \mathcal{P}_{up}(k')$.

Lemma 18 (Logical Time Bounds). *For any obedient process $p \in \mathcal{P}_{up}(k)$, $k \geq 1$, and any t' with $\sigma_p^{k-2} \leq t' < \sigma_p^{k-1}$,*

$$\begin{aligned} C_{\max}(t') &\leq C_p(t') + D_q = k - 1 + D_q && (7.5)\\ C_{\max}(\sigma_p^{k-1}) &\leq C_p(\sigma_p^{k-1}) + D_q - 1 = k - 1 + D_q && (7.6)\\ \textit{with} & \\ D_q &= \lfloor \Omega + 2 \rfloor. && (7.7) \end{aligned}$$

Proof. If $C_{\max}$ never reaches $k+1$, (7.5) and (7.6) hold trivially since $D_q \geq 1$. Hence, suppose that $C_{\max}$ reaches $k+1$ at time $\sigma^k_{first} < \infty$. Let $t = \sigma_p^{k-1}$ denote the time when p sets its clock to k. In order to compute $C_{\max}(t)$, we use Lemma 4 and our knowledge of $C_{\max}(t-\varepsilon) \leq C_{\max}(\sigma^k_{first}) = k+1$ due to monotonicity of $C_{\max}$ in conjunction with (7.3): From Lemma 4, we infer that $C_{\max}(t) \leq \lfloor \frac{t-(t-\varepsilon)}{\tau_f} \rfloor + I_\sigma(t-\varepsilon) + C_{\max}(t-\varepsilon)$. If $\sigma^k_{first} = t - \varepsilon$ such that $t-\varepsilon$ is synchronized with $C_{\max}$, then $C_{\max}(t-\varepsilon) = k+1$ and $I_\sigma(t-\varepsilon) = 0$ according to Definition 6. If, on the other hand, $\sigma^k_{first} > t - \varepsilon$, then $C_{\max}(t-\varepsilon) = k$ and $I_\sigma(t-\varepsilon) = 1$. In both cases, $I_\sigma(t-\varepsilon) + C_{\max}(t-\varepsilon) = k+1$ and hence $C_{\max}(t) \leq \lfloor \frac{\varepsilon}{\tau_f} \rfloor + k + 1 \leq \lfloor \Omega + 2 \rfloor + k - 1$. As $C_{\max}(t') \leq C_{\max}(t)$ since $t' < t$ and $C_p(t') = k-1$, (7.5) follows. If we increase t' until $t' = t = \sigma_p^{k-1}$, then $C_p(t')$ increases from $k-1$ to k and thus reduces the difference to $C_{\max}$ by 1, as asserted in (7.6). □

The following Lemma 19 is a simple corollary of Lemma 18. It immediately leads to the precision bound given in Theorem 19 below.

Lemma 19 (Precision Bound). *Let t be any real-time where $C_{\max}(t) = K \geq 1$, and p be an obedient process with $C_p(t) = k$ for some $0 \leq k \leq K$. If $p \in \mathcal{P}_{up}(k+1)$, then $k \geq K - D_q$. If $t = \sigma_p^{k-1}$ and $p \in \mathcal{P}_{up}(k)$, then $k \geq K - D_q + 1$.*

Proof. Inequality (7.5) in Lemma 18 reveals that $C_{\max}(t') \leq k + D_q$ for all $\sigma_p^{k-1} \leq t' < \sigma_p^k$ since $p \in \mathcal{P}_{up}(k+1)$ by assumption. Given that $C_p(t) = k$ implies $t < \sigma_p^k$ here, we can choose $t' = t$ such that $K = C_{\max}(t) \leq k + D_q$ as asserted. If, on the other hand, $t = \sigma_p^{k-1}$, then (7.6) applies and hence $K = C_{\max}(t) \leq k + D_q - 1$. □

Theorem 19 (Bulk Precision). *Consider a system of* $n \geq f_\ell^s + 2f_\ell^{ra} + 2f_\ell^r + 3f_a + 2f_s + 2f_o + f_c + 1$ *processes running the algorithm of Figure 7.4. For any* $K \geq 1$ *and all real-times* $t \geq \sigma_{first}^{K-1}$*, all obedient processes* $p \in \mathcal{P}_{up}(k_p + 1)$ *with* $k_p = C_p(\sigma_{first}^{K-1})$ *maintain a mutual precision of* $D_q = \lfloor \Omega + 2 \rfloor$.

Proof. Let $K' \geq K$ be such that $\sigma_{first}^{K'-1} \leq t < \sigma_{first}^{K'}$ and hence $C_{\max}(t) = K'$. Applying Lemma 19 reveals that all obedient processes $p \in \mathcal{P}_{up}(k'_p + 1)$ with $k'_p = C_p(t)$ satisfy $K' - k'_p \leq D_q$ and are hence within precision D_q. Since $p \in \mathcal{P}_{up}(k_p + 1)$ implies $p \in \mathcal{P}_{up}(k'_p + 1)$ for any $k'_p \geq k_p$, our lemma follows. □

With those preparations, we are now ready for computing the precision of our algorithm during early and (the beginning of) degraded mode: Since no obedient process can advance its clock from 0 to 1 as long as less than $n - f_\ell^s - f_\ell^r - f_a - f_s - f_o - f_c$ processes have completed booting, see Figure 7.4 (`line 13`), the system remains in early mode until the $n - f_\ell^s - f_\ell^r - f_a - f_s - f_o - f_c$-th process p (faulty ones included) completes booting, at some time t_{up}. Progress of any obedient process—and hence the transition to degraded mode—is possible only after time $t_{up} + \tau^-$, since p's *(round* 0*)*, as well as the assistance of all faulty processes, is needed for this purpose. Hence, degraded mode starts at some time $\sigma_{first}^0 \geq t_{up} + \tau^-$, with some specific number $n - f_\ell^s - f_\ell^r - f_a - f_s - f_o - f_c \leq n_{up}(1) \leq n$ of processes that have completed booting soon enough to be able to receive each other's tick 1 messages. Note that all these $n_{up}(1)$ processes have clock value 0 or 1, and are hence a priori synchronized to each other within precision 1 during early mode.

Setting $K = 1$ in Theorem 19 immediately yields the precision of our algorithm in a system with $n \geq f_\ell^s + 2f_\ell^{ra} + 2f_\ell^r + 3f_a + 2f_s + 2f_o + f_c + 1$ processes, where an arbitrary but fixed number $n_{up}(1) \leq n$ of processes start in degraded mode. It shows that all obedient processes among the $n_{up}(1)$ ones are synchronized to within D_q for all times.

Theorem 20 (Precision of Early Starters). *Consider a system of* $n \geq f_\ell^s + 2f_\ell^{ra} + 2f_\ell^r + 3f_a + 2f_s + 2f_o + f_c + 1$ *processes with a fixed number* $n_{up}(1) = |\mathcal{P}_{up}(1)| \leq n$ *of processes that completed booting by time* σ_{first}^0 *and are hence ready to receive all their tick 1 messages. The obedient processes* $p \in \mathcal{P}_{up}(1)$ *are synchronized to each other within* $D_q = \lfloor \Omega + 2 \rfloor$ *during whole system lifetime.*

Proof. Since all obedient processes in $\mathcal{P}_{up}(1)$ have clock value 0 or 1, they are obviously synchronized within precision $1 \leq D_q$ up to time σ_{first}^0. From time σ_{first}^0 on, Theorem 19 secures precision D_q due to the fact that all obedient processes p with $C_p(\sigma_{first}^0) = k \leq 1$ are up since time $\sigma_{first}^0 \leq \sigma_{first}^k$ according to our assumption $p \in \mathcal{P}_{up}(1)$. □

7.5 From Degraded to Normal Mode

Theorem 20 can be applied to all obedient processes that have completed booting by the beginning of degraded mode. It shows that all those early starters remain synchronized to each other within D_q. The problem is, however, that additional processes might complete booting during degraded mode, after the set of early starters has made some progress. As late processes start with a clock value of 0, they are clearly not synchronized right from the start. This does not matter, since passive processes need not satisfy precision (π). Unfortunately, however, even active late starters could compute their first clock values from very little information on the system state. Although they will eventually also reach precision D_q, this results in a considerably larger precision D_{boot} during a short period of time after becoming active. Since no process can determine locally whether it is already within D_q, it is D_{boot} that determines our algorithm's worst case precision. This fact is essential when considering other algorithms during booting, later in Section 8: A priori knowledge of D_{boot} will be used to derive a local estimation of $C_{max}(t)$ when some process becomes active at time t.

Besides the larger precision D_{boot}, late starters introduce another disadvantageous property: In order to guarantee that all correct processes become active and synchronized within bounded time, we have to guarantee a sufficiently large "leader group" of correct processes. Therefore, our algorithm requires an increased number of processes for some restricted failure modes and for tolerating send link failures[11] in *join*-messages: It will turn out that $f_s + f_c + f_\ell^s$ additional processors are required here, i.e., that we need a system with $n \geq 2f_\ell^s + 2f_\ell^{ra} + 2f_\ell^r + 3f_a + 3f_s + 2f_o + 2f_c + 1$ processes. Unlike in Section 7.4, symmetric process failures are hence as severe as asymmetric ones, and crash failures are as severe as our omission failures in presence of late starters. Note that this can be explained by the fact that (receiver) processes boot at unpredictable times, which turns otherwise consistently perceived process failures into inconsistently perceived ones.

In order to compute D_{boot}, we first give a bound on real-time t_{sync} when an obedient late starter process p reaches a local clock value $C_p(t_{sync})$ within precision D_q, such that $C_{\max}(t) - C_p(t) \leq D_q$ for any time $t \geq t_{sync}$. Time t_{sync} hence delimits the period where p may have the worst case precision D_{boot}. It must be noted, however, that t_{sync} depends upon progress of $C_{\max}$ here (which is not guaranteed during booting).

Lemma 20 (First Synchronization). *Let p be an obedient process that gets up at time t_{up} and let $k = C_{\max}(t_{up} - \tau^- - \epsilon) + 1 \geq 1$ for any infinitesimally small ϵ. Then, $p \in \mathcal{P}_{up}(k)$, $C_p(t_{sync}) = k$ where $t_{sync} \leq \sigma^k_{first} + \varepsilon$, and*

$$C_{\max}(t) - C_p(t) \quad \leq \quad D_q \tag{7.8}$$

for all times $t \geq t_{sync} = \sigma_p^{k-1}$. Moreover,

$$C_{\max}(t_{sync} - \epsilon) \leq C_{\max}(t_{up} - \tau^- - \epsilon) + \lfloor \Omega + 2 \rfloor. \tag{7.9}$$

[11] Send link failures ($f_\ell^s > 0$) must be considered explicitly only in Theorem 22 and Theorem 24; all other results hold for $f_\ell^s = 0$ (despite of $f_\ell^r > 0$) as well.

Proof. Our choice of k ensures $\sigma^{k-1}_{first} \geq t_{up} - \tau^-$ and hence $t_{up} \leq \sigma^{k-1}_{first} + \tau^-$, such that $p \in \mathcal{P}_{up}(k)$ by Lemma 16. Hence, Lemma 17 holds and (7.3) ensures $t_{sync} = \sigma^{k-1}_p \leq \sigma^k_{first} + \varepsilon$ and hence $C_p(t) \geq k$ for all $t \geq t_{sync}$. Lemma 19 guarantees that p is within precision D_q for all $t \geq t_{sync}$ as asserted in (7.8).

In oder to confirm (7.9), we abbreviate $t' = \sigma^k_{first}$ such that $C_{\max}(t') = C_{\max}(t_{up} - \tau^- - \epsilon) + 2 = k + 1$ and $I_\sigma(t') = 0$ according to Definition 6. From Lemma 4, it follows that $C_{\max}(t_{sync} - \epsilon) \leq C_{\max}(t' + \varepsilon - \epsilon) \leq C_{\max}(t') + \lfloor \frac{(t'+\varepsilon-\epsilon)-t'}{\tau_f} \rfloor + I_\sigma(t') \leq C_{\max}(t_{up} - \tau^- - \epsilon) + 2 + \lfloor \Omega \rfloor = C_{\max}(t_{up} - \tau^- - \epsilon) + \lfloor \Omega + 2 \rfloor$ as asserted. □

Next we have to consider the worst case quality of the information that a passive process p might use to change to active. We start with Lemma 21, which reveals that messages from some common obedient process r are required for both advancing $C_{\max}$ and changing p to active (`line 13`). Note carefully, however, that it need not be the same message from r since $h = 1$.

Lemma 21 (Common Process). *For* $n \geq 2f^s_\ell + 2f^{ra}_\ell + 2f^r_\ell + 3f_a + 3f_s + 2f_o + 2f_c + 1$, *any two obedient processes p and q that execute* `line 13` *in the algorithm of Figure 7.4 use messages from at least one common obedient process r.*

Proof. Let $f'_a \leq f_a$ and $f'_s \leq f_s$ be the actual number of arbitrary and symmetric faulty processes, respectively. Both processes p and q must have got at least $n - f^s_\ell - f^r_\ell - f_a - f_s - f_o - f_c$ *(round k_p)* and *(round k_q)* perceptions, respectively. At most $f'_a + f'_s + f^{ra}_\ell$ of those could have a malign faulty origin (see Lemma 12), such that $m = n - f^s_\ell - f^r_\ell - f_a - f_s - f_o - f_c - f'_a - f'_s - f^{ra}_\ell$ is the minimal number of perceptions obtained from obedient processes at either p and q. The total number of obedient processes in the system is at most $n - f'_a - f'_s$. Since

$$\begin{aligned} 2m - n + f'_a + f'_s &= n - 2f^s_\ell - 2f^r_\ell - 2f_a - 2f_s - 2f_o \\ &\quad -2f_c - f'_a - f'_s - 2f^{ra}_\ell \\ &\geq n - 2f^s_\ell - 2f^r_\ell - 2f^{ra}_\ell - 3f_a - 3f_s \\ &\quad -2f_o - 2f_c \\ &\geq 1, \end{aligned} \tag{7.10}$$

the pigeonhole principle reveals that at least one of p and q's *(round)*-perceptions must originate from the same obedient process r (although it need not be the same message from r that generated those perceptions at p and q). □

We are now ready for computing D_{boot}. The worst situation with respect to $C_{\max}$ occurs when a late starter p switches to active shortly after getting up, such that is has very few information on the system state, and that $C_{\max}$ progresses at maximum rate until time t_{sync}, where p is guaranteed to be within D_q. The worst case precision D_{boot} then occurs shortly before t_{sync}.

Theorem 21 (Precision). *For* $n \geq 2f^s_\ell + 2f^{ra}_\ell + 2f^r_\ell + 3f_a + 3f_s + 2f_o + 2f_c + 1$, *the algorithm of Figure 7.4 achieves the precision property (π) during whole system life-time with* $D_{boot} = \lfloor 2\Omega + 4 \rfloor$.

Proof. Let t_{up} be the time when late starting process p gets up. Assume that process p changes to active at time $t_1 \geq t_{up}$, with a clock value of $k+1$, based on a *(round* $k+1$*)* message received at time $t_1' \leq t_1$ that was sent by the common obedient process r guaranteed by Lemma 21. This *(round* $k+1$*)* message is "interpreted" as *(round* k*)* at p, which is legitimate due to the history size $h = 1$. In addition, consider time $t_2 = \sigma_{first}^{k+1}$, when the first obedient process q increases $C_{\max}$ to $k+2$. According to Lemma 21, a *(round* $k+1$*)* message from the same process r must be present at q at time $t_2' \leq t_2$. We must distinguish two cases here: The *(round* $k+1$*)* message received at p could be the result of r's "regular" *(round* $k+1$*)* broadcast, or r's response to p's *join*-message.

If *(round* $k+1$*)* at p and q are from r's broadcast, $|t_1' - t_2'| \leq \varepsilon$. The worst case precision occurs when r sent *(round* $k+1$*)* as early as possible, such that $C_{\max}$ can progress as much as possible before p gets the associated messages (reception of these messages would require p to catch up to a clock value $> k+1$). The worst setting is hence $t_1' = t_{up}$ and $t_2 = t_2' = \sigma_{first}^{k+1} = t_{up} - \varepsilon$. Since $t_2 = \sigma_{first}^{k+1}$, t_2 is synchronized with $C_{\max}$ and hence $I_\sigma(t_2) = 0$ by Definition 6. If $t_2 \geq t_{up} - \tau^-$ it follows that $C_{\max}(t_{up} - \tau^- - \epsilon) \leq C_{\max}(t_2) - 1 = k+1$ which is smaller than the case where $t_2 < t_{up} - \tau^-$ for which Lemma 4 hence reveals

$$\begin{aligned} C_{\max}(t_{up} - \tau^- - \epsilon) &\leq C_{\max}(t_2) + \left\lfloor \frac{(t_{up} - \tau^- - \epsilon) - (t_2)}{\tau_f} \right\rfloor + I_\sigma(t_2) \\ &= k + 2 + \left\lfloor \frac{\varepsilon - \tau^- - \epsilon}{\tau_f} \right\rfloor \\ &\leq \lfloor \Omega \rfloor + k + 2 \end{aligned} \tag{7.11}$$

for any infinitesimally small ϵ here.

If, alternatively, p gets its *(round* $k+1$*)* as a response to its *join*-message, process r must not have caught up to some clock value $> k+1$. Hence, if t_0 denotes the real-time when *join* arrives at r, we must have $t_0 < \sigma_r^{k+1} \leq \sigma_{first}^{k+2} + \varepsilon$ (and hence $t_1' < \sigma_{first}^{k+2} + \varepsilon + \tau^+$ since the response message may travel at most τ^+). After all, $r \in \mathcal{P}_{up}(k+2)$ since r helped to advance $C_{\max}$ to $k+2$, such that (7.3) in Lemma 17 implies $\sigma_r^{k+1} \leq \sigma_{first}^{k+2} + \varepsilon$. Allowing *join* to travel only τ^- in order to maximize t_{up} in our worst case setting, this implies $t_{up} < \sigma_{first}^{k+2} + \varepsilon - \tau^-$. Abbreviating $t_3 = \sigma_{first}^{k+2}$, this implies

$$\begin{aligned} C_{\max}(t_{up} - \tau^- - \epsilon) &\leq C_{\max}(t_3) + \left\lfloor \frac{\varepsilon - 2\tau^- - \epsilon}{\tau_f} \right\rfloor \\ &= k + 3 + \left\lfloor \frac{\tau^+ - 3\tau^- - \epsilon}{\tau_f} \right\rfloor \\ &\leq \lfloor \Omega \rfloor + k + 3 \end{aligned} \tag{7.12}$$

here. Comparison with (7.11) reveals that this constitutes the worst case situation.
We are still free to choose t_1 to complete our worst case setting for D_{boot}. Note that p could switch to active based on messages with malign origin or sent by passive

processes, which means that $t_1 \geq t'_1$ can be arbitrary. We assume that this happens immediately before t_{sync}, the time when p is guaranteed to be within precision D_q according to Lemma 20. Recalling (7.9), the required bound for D_{boot} is hence implied by $C_{\max}(t_{sync}-\epsilon) \leq C_{\max}(t_{up}-\tau^- -\epsilon)+\lfloor\Omega+2\rfloor \leq \lfloor\Omega\rfloor+k+3+\lfloor\Omega+2\rfloor \leq \lfloor 2\Omega+4\rfloor+k+1$, such that $C_p(t_1) = k+1 \geq C_{\max}(t_{sync}-\epsilon) - \lfloor 2\Omega+4\rfloor$ as asserted. □

Apart from precision, it is not difficult to show that all active processes also satisfy the upper envelope bound of the accuracy property (α):

Lemma 22 (Upper Envelope Bound). *For $n \geq 2f_\ell^s + 2f_\ell^{ra} + 2f_\ell^r + 3f_a + 3f_s + 2f_o + 2f_c + 1$, all obedient processes p that are active by time t_1 satisfy $C_p(t_2) - C_p(t_1) < \frac{t_2-t_1}{2\tau^-} + D_{boot} + 1$ for all times $t_2 \geq t_1$.*

Proof. From Theorem 21, it follows that $C_{\max}(t) - D_{boot} \leq C_p(t) \leq C_{\max}(t)$ at all times $t \geq t_1$. Thus, $C_p(t_2) - C_p(t_1) \leq C_{\max}(t_2) - C_{\max}(t_1) + D_{boot}$. Applying Lemma 4, the statement of our lemma follows immediately. □

Theorem 21 reveals that late starters can be guaranteed to maintain precision when they switch to active mode. What still needs to be shown, however, is that this switch to active occurs eventually. In order to prove this, we first show in Lemma 23 that, at any time t, at least $f_\ell^s + f_\ell^{ra} + f_\ell^r + f_a + f_s + 1$ correct processes have clock values of $C_{\max}(t)$ or $C_{\max}(t) - 1$ and emitted corresponding *echo*-messages by time t.

Lemma 23 (Frontier Processes). *Consider any time t with $k = C_{\max}(t) > 0$. In a system with $n \geq 2f_\ell^s + 2f_\ell^{ra} + 2f_\ell^r + 3f_a + 3f_s + 2f_o + 2f_c + 1$ processes, the algorithm of Figure 7.4 ensures that there are at least $f_\ell^s + f_\ell^{ra} + f_\ell^r + f_a + f_s + 1$ correct processes with local clock values k or $k-1$ which have sent (round k) or (round $k-1$).*

Proof. Let p be the first obedient process that sets its clock to the maximum at time t, such that $C_p(t) = C_{\max}(t) = k$. According to Lemma 15, this happens via `line 13`, so p satisfies $|\mathcal{V}_p^{(k-1)}(t)| \geq n - f_\ell^s - f_\ell^r - f_a - f_s - f_o - f_c \geq f_\ell^s + 2f_\ell^{ra} + f_\ell^r + 2f_a + 2f_s + f_o + f_c + 1$. At least $f_\ell^s + f_\ell^{ra} + f_\ell^r + f_a + f_s + 1$ of those originate from correct processes that sent either *(round $k-1$)* or *(round k)*, recall the history size $h = 1$. Since correct processes never send *(round)*messages for ticks larger than their local clock values, they must all have a clock value of k or $k-1$ at time t. □

The following major Theorem 22 implies a bound on the time t_{sync} when a late starting process p gets synchronized within D_q, which—unlike Lemma 20—does not depend upon progress of $C_{\max}$. The problem with Lemma 20 is that one cannot a priori bound the difference between t_{up} and $t_{sync} \leq \sigma_{first}^k + \varepsilon$, since σ_{first}^k can happen at any time in the future. In this case, however, p actually gets synchronized when the *join*-protocol terminates, i.e., by time $t_{up} + 2\tau^+$: Since there is no further progress of $C_{\max}$ here, the responses to *join* recover all missed tick $k-1$ or $k-2$ messages. Having received all the responses, the late starter p can hence catch up to $k-2$ as if it had been up right from time σ_{first}^{k-3} on, i.e., as if $p \in \mathcal{P}_{up}(k-2)$. A similar argument applies if $C_{\max}$ progresses to k during the *join*-protocol. It is important to note, however, that p need not be active at time t_{sync}. That is, p is already well-synchronized at t_{sync} but does not know this fact.

Theorem 22 (First Synchronization via Join). *Let p be an obedient process that gets up at time $t_{up} \geq \sigma^0_{first}$ in a system with $n \geq 2f^s_\ell + 2f^{ra}_\ell + 2f^r_\ell + 3f_a + 3f_s + 2f_o + 2f_c + 1$ processes running the algorithm of Figure 7.4, and assume that $k = C_{\max}(t_{up} - \tau^- - \epsilon) + 1 \geq 1$ for any infinitesimally small ϵ. Then, there is some $t_{sync} \leq t_{up} + 2\tau^+$ with $\ell = C_p(t_{sync}) \geq k - 2$, such that $p \in \mathcal{P}_{up}(\min\{\ell, k\})$ and*

$$C_{\max}(t) - C_p(t) \quad \leq \quad D_q \tag{7.13}$$

for all times $t \geq t_{sync}$.

Proof. Since $k = C_{\max}(t_{up} - \tau^- - \epsilon) + 1 \geq 1$, it follows that $\sigma^{k-1}_{first} \geq t_{up} - \tau^-$ and hence $p \in \mathcal{P}_{up}(k)$ by Lemma 16. Consequently, k is the first tick that is observed comprehensively by p. We will show that p gets sufficiently many messages to set its clock at least to $\ell \geq k - 2$ at some time t_{sync} with $\sigma^{\ell-1}_p \leq t_{sync} \leq t_{up} + 2\tau^+$, and that $p \in \mathcal{P}_{up}(\ell)$ and (7.13) hold from this time on.

Since p sends its *join* message at time t_{up}, any obedient process receives it within $\boldsymbol{I} = [t_{up} + \tau^-, t_{up} + \tau^+]$. By time $t_{up} + 2\tau^+$, all the responses from correct processes must have arrived at p. We have to distinguish 3 cases:

Case (a): If $\sigma^k_{first} \leq t_{up} + \tau^+$, Lemma 20 reveals $t_{sync} \leq t_{up} + \tau^+ + \varepsilon \leq t_{up} + 2\tau^+$ and hence confirms our lemma for $\ell = k$. So let us assume that $\sigma^k_{first} > t_{up} + \tau^+$, which implies that only σ^{k-1}_{first} could occur within $\boldsymbol{I}$.

Case (b): If $\sigma^{k-1}_{first} \notin \boldsymbol{I}$, the value of $C_{\max}$ does not increase within $\boldsymbol{I}$, i.e., $k' = C_{\max}(t_{up} + \tau^-) = C_{\max}(t_{up} + \tau^+)$ with $k' = k$ or $k' = k - 1$, depending upon whether (b.1) $\sigma^{k-1}_{first} < t_{up} + \tau^-$ or (b.2) $\sigma^{k-1}_{first} > t_{up} + \tau^+$ (the latter is only possible for $k \geq 2$ due to $t_{up} \geq \sigma^0_{first}$). Lemma 23 assures that there is a set $\mathcal{P}$ of at least $f^s_\ell + f^{ra}_\ell + f^r_\ell + f_a + f_s + 1$ correct processes at time $t_{up} + \tau^-$ that last sent *(round k')* or *(round $k' - 1$)*. Since correct processes send *echo*-messages with increasing index only, and k' is the maximum clock value throughout $\boldsymbol{I}$ and hence the maximum possible index, all the processes in $\mathcal{P}$ last sent *(round k')* or *(round $k' - 1$)* at any time $t \in \boldsymbol{I}$. Since at most f^s_ℓ of the processes in $\mathcal{P}$ may have lost p's *join* due to send link failures, there are at least $f^{ra}_\ell + f^r_\ell + f_a + f_s + 1$ *(round k')* or *(round $k' - 1$)* among the responses on p's *join*-message. Recalling the at most f^r_ℓ receive link failures, it follows that there is some time $\sigma^{k'-2}_p \leq t_{sync} \leq t_{up} + 2\tau^+$ with $|\mathcal{V}^{(k'-1)}_p(t_{sync})| \geq f^{ra}_\ell + f_a + f_s + 1$. This triggers `line 6` that causes $C_p(t_{sync}) \geq k' - 1$, if this has not happened earlier. Moreover, by time t_{sync}, we can be sure that the responses to *join* recovered all but at most f^s_ℓ tick $k' - 1$ messages that might have been missed due to late booting. Hence, at time t_{sync}, the situation at p is equivalent to $p \in \mathcal{P}_{up}(k' - 1)$. Note that the up to f^s_ℓ missing *(round $k' - 1$)* or *(round k')* originating from lost *join* messages do not matter for the pivotal quasi simultaneity (Q) property, since our n is larger by f^s_ℓ than the n of Theorem 18. Lemma 19 thus guarantees that p is within precision D_q for all times $t \geq t_{sync}$ and hence (7.13).

Case (c): Finally, if $\sigma^{k-1}_{first} \in \boldsymbol{I}$, the value of $C_{\max}$ increases from $k-1$ to k somewhere within interval $\boldsymbol{I}$. Consider the at least $f^s_\ell + f^{ra}_\ell + f^r_\ell + f_a + f_s + 1$ correct processes that last sent *(round k)* or *(round $k - 1$)* according to Lemma 23 at time σ^{k-1}_{first}. If p's *join*-message reached one of those processes q at some time $t \geq \sigma^{k-1}_{first}$, its response

message is guaranteed to be *(round $k-1$)* or *(round k)* by the same reasoning as before. If, on the other hand, p's *join*-message reached q at some time $t < \sigma_{first}^{k-1}$, there are two possibilities: If the index of the last sent *echo*-message of q was already $k-1$ at time t, the response message is *(round $k-1$)* as required. Otherwise, the response message is some *(round $k-x$)* with $x \geq 2$. However, by time σ_{first}^{k-1}, process q must have changed its state such that it last sent *(round k)* or *(round $k-1$)* according to Lemma 14, which means that this message must already be in transit and reaches p by $\sigma_{first}^{k-1} + \tau^+ \leq t_{up} + 2\tau^+$. Like above, it hence follows that there is some time $\sigma_p^{k-2} \leq t_{sync} \leq t_{up} + 2\tau^+$ with $|\mathcal{V}_p^{(k-1)}(t_{sync})| \geq f_\ell^{ra} + f_a + f_s + 1$, which triggers `line 6` and causes $C_p(t') \geq k-1$, if this has not happened earlier. By the very same reasoning as in case (b) above, we can claim that the *join*-protocol established a situation equivalent to $p \in \mathcal{P}_{up}(k-1)$, and that hence Lemma 19 implies (7.13) as required. This eventually completes the proof of Lemma 22. □

Equipped with Theorem 22, it is not difficult to bound the time it takes for all obedient processes to become active after the $n - f_a - f_s - f_o - f_c$-th correct process got up. We need some additional properties related to Theorem 18 for this purpose, however, which apply only when at least $n - f_\ell^s - f_a - f_s - f_o - f_c$ processes are up in a system with $n \geq f_\ell^s + 2f_\ell^{ra} + 2f_\ell^r + 3f_a + 3f_s + 2f_o + 2f_c + 1$ processes.

Theorem 23 (Clock Synchronization Properties). *In a system with $n \geq f_\ell^s + 2f_\ell^{ra} + 2f_\ell^r + 3f_a + 3f_s + 2f_o + 2f_c + 1$ processes, the algorithm from Figure 7.4 achieves*

(P) Uniform Progress. *If at least $n - f_\ell^s - f_a - f_s - f_o - f_c$ correct processes set their clocks to $k \geq 0$ by time t, then every obedient process $\in \mathcal{P}_{up}(k)$ sets its clock at least to $k+1$ and becomes active by time $t + \tau^+$.*

(S) Uniform Simultaneity. *Suppose there are at least $n - f_\ell^s - f_a - f_s - f_o - f_c$ correct processes among the processes $\in \mathcal{P}_{up}(k-1)$. If any obedient process sets its clock to $k > 0$ at time t, then every obedient process $\in \mathcal{P}_{up}(k-1)$ also sets its clock at least to k and becomes active by time $t + \tau^+ + \varepsilon$.*

Proof. Uniform Progress. According to Lemma 14, all of the at least $n - f_\ell^s - f_a - f_s - f_o - f_c \geq 2f_\ell^{ra} + 2f_\ell^r + 2f_a + 2f_s + f_o + f_c + 1$ correct processes must have sent *(round k)* by time t. Taking into account the at most f_ℓ^r receive link failures, the *(round k)* perception vector of every obedient process $p \in \mathcal{P}_{up}(k)$ at time $t + \tau^+$ hence satisfies $|\mathcal{V}_p^{k'}(t + \tau^+)| \geq n - f_\ell^s - f_\ell^r - f_a - f_s - f_o - f_c$. Consequently, p sets its clock to $k+1$ by `line 13` in Figure 7.4, if it had not already done so. Hence, switching to active mode is guaranteed to happen in this case as well.

Uniform Simultaneity. According to (Q) all correct processes set their clock to $k-1$ by $t + \varepsilon$. By (P) all processes set their clock to k by time $t + \tau^+ + \varepsilon$. □

Now we are ready for our Theorem 24, which bounds the time it takes for all obedient processes to become active after the last correct process got up, and shows that the system is guaranteed to make progress afterward.

Theorem 24 (Initialization Time). *Let $t_{up} \geq \sigma^0_{first}$, with $k = C_{\max}(t_{up} - \tau^- - \epsilon) + 1 \geq 1$ for any infinitesimally small ϵ, be the time when the $n - f_a - f_s - f_o - f_c$-th correct process gets up in a system of $n \geq 2f^s_\ell + 2f^{ra}_\ell + 2f^r_\ell + 3f_a + 3f_s + 2f_o + 2f_c + 1$ processes running the algorithm of Figure 7.4. All obedient processes that got up by time t_{up} switch to active by time $t_{up} + \Delta_{init}$ and progress to at least k by time $t_{up} + \Delta_{init} + \tau^+$, where $\Delta_{init} = 2\tau^+ + \varepsilon$.*

Proof. Let p be the $n - f_a - f_s - f_o - f_c$-th correct process that gets up at t_{up}. Theorem 22 ensures that it gets synchronized with $C_{\max}$—and hence with all other processes that are up at t_{up}—within D_q at time $t_{sync} \leq t_{up} + 2\tau^+$ and satisfies $\ell_p = C_p(t_{sync}) \geq k - 2$ as well as $p \in \mathcal{P}_{up}(\min\{\ell_p, k\})$. Obviously, $q \in \mathcal{P}_{up}(\min\{\ell_q, k\})$ and precision D_q also holds for obedient processes q that booted earlier, including the early starters. Abbreviating $t' = t_{sync}$ and $k' = C_{\max}(t') \geq k - 1$, we will first show that every obedient process q sets its clock to at least $k' \geq k - 1$ and becomes active by time $t' + \tau^+ + \varepsilon \leq t_{up} + \Delta_{init}$. If already $C_q(t') \geq k'$, nothing remains to be done. In the other case $C_q(t') \leq k' - 1$, we can apply simultaneity (S) in Theorem 23 since $q \in \mathcal{P}_{up}(k' - 1)$ due to $q \in \mathcal{P}_{up}(\min\{\ell_q, k\})$ and $\ell_q \leq C_q(t') \leq k' - 1$. Hence, q must set its clock to at least k' and switch to active by time $t' + \tau^+ + \varepsilon$ as asserted.

Using progress (P), it is easy to show that every obedient process q reaches at least clock value $k' + 1 \geq k$ by time $t' + 2\tau^+ + \varepsilon$. Every process q has already reached clock value k', so obviously $q \in \mathcal{P}_{up}(k')$. Hence, q sets its clock to at least $k' + 1$ within another τ^+. This eventually completes the proof of Theorem 24. □

We still have to consider very late processes, which complete booting after the $n - f_a - f_s - f_o - f_c$-th correct one. The following Theorem 25 gives a bound upon the time until such a process becomes active. Since we have already guaranteed progress here, this is equivalent to integrating processes into running systems.

Theorem 25 (Integration Time). *Suppose that some obedient process gets up at time t_{up} after the $n - f_a - f_s - f_o - f_c$-th correct one. Then, p switches to active by time $t_{up} + \Delta_{init}$ with $\Delta_{init} = 2\tau^+ + \varepsilon$.*

Proof. The proof of Theorem 24 does not depend upon p being the $n - f_a - f_s - f_o - f_c$-th correct process, but could be any obedient process booting after the $n - f_a - f_s - f_o - f_c$-th correct one. Hence, it can be re-used literally here. □

Since this section's clock synchronization algorithm ensures uniform versions of (P), (U), (Q), and (S) during normal mode, it also satisfies (π) and (α) as given in Theorem 5 and Theorem 6, respectively.

7.6 Related Work

Clock synchronization in distributed systems is a very well researched field, see [30, 93, 82, 92, 84, 73] for an overview. Still, there are only a few papers [95, 73, 70, 57, 96] known to us that deal with initial synchronization, mostly in the context of integrating a new process in an already running system. For initialization it is often assumed that all correct processes are up and listening to the network when the algorithm is started [95, 70]. In systems where processes boot at unpredictable times, this assumption is too strong.

Some solutions for booting exist for very specific architectures: System startup of TTP—viewed as change from asynchronous to synchronous operation—has been investigated in [96]. Initial clock synchronization for the MAFT architecture [57] has been solved, but under stronger system assumptions: A priori assumptions on message transmission delay and local timers are used there to construct a sufficiently large listen window. Termination is achieved by Byzantine Agreement, which, however, requires $2f + 1$ correct processes to be up and running. This cannot always be guaranteed during startup, however. Our goal is minimizing the number of such a priori assumptions. Still we do not know of any approach that could be compared to ours [106, 109] with respect to partial synchrony in conjunction with initially down correct processes.

Our clock synchronization algorithm has *graceful degradation* [72] during system booting. This prevents Byzantine processes from corrupting the system state during the startup phase where more than one third of the running processes may be Byzantine. Mahaney and Schneider [72] introduced synchronous approximate agreement algorithms which provide graceful degradation when between $1/3$ and $2/3$ of the processes are faulty. We reach the same bounds.

If the algorithm would have no graceful degradation, the system could be forced into arbitrary states and the solution for initialization must be *self-stabilizing* [28]. Most self-stabilizing clock synchronization algorithms [34, 79, 6] do not stabilize if some processes remain faulty during the whole execution. Still, there exist self-stabilizing Byzantine clock synchronization algorithms [25, 33] as well. They are an overkill for solving the booting problem, however. None of the existing solutions guarantees constant stabilization times like we do for booting. Moreover, we consider a weaker (time free) system model.

Our results are also related to the crash recovery model [3], where processes crash and recover arbitrarily during the execution of a consensus algorithm. Similar work was conducted in the context of clock synchronization [14]. We, however, consider Byzantine processes and more than $n/2$ "crashed" (actually, "initially dead") processes during startup. This exceeds the bounds used in [3, 14].

Chapter 8

Booting Θ-Algorithms

The previous Section 7 presented a clock synchronization solution that guarantees some of its properties during system booting. In order to cope with the reduced number of participating processes it was required to relax the accuracy requirement (α) during the booting phase. In this section we will discuss how to adapt the algorithms from Section 6 in order to cope with booting as well. Again, we will see that some properties that can be achieved easily when all correct processes are always up must be relaxed here.

In order to keep the presentation concise we will again use restricted fault models in this section. In particular we do not consider link faults, hence $f_\ell^{ra} = 0$, $f_\ell^r = 0$ and $f_\ell^s = 0$. The types of process failures considered depend on the problem (similar to Section 6).

8.1 Eventually Perfect Failure Detector

Recalling the semantics of the perfect failure detector $\mathcal{P}$, the properties of our clock synchronization algorithm suggest two approaches for adding system startup to the FD implementation of Section 6.2. First, we noted already in Section 7 that our algorithm maintains some precision $D_{boot} > D_{max}$ during whole system lifetime. Hence, if we based $\Xi_{\mathcal{P}}$ upon D_{boot}, most of the proof of Theorem 9 would apply also during system startup: (SC) is guaranteed, since progress of clock values is eventually guaranteed, namely, when normal mode is entered. The major part of the proof of (SA) is also valid, provided that D_{max} is replaced by D_{boot}.

There is one remaining problem with this approach, however: During system startup, the resulting algorithm could suspect a correct process that simply had not started yet. When this process eventually starts, it is of course removed from the list of suspects – but this must not happen in case of a perfect failure detector. Note that not suspecting processes that never send any message until transition to normal mode does not work either, since a process cannot reliably detect when the transition to normal mode happens and could hence "overlook" a crashed processor. Consequently, unless the perfect FD specification is extended by the notion of "not yet started" processes, there is no hope of implementing $\mathcal{P}$ also during system startup.

The alternative is to accept degraded failure detection properties during system startup: We will show below that the failure detector of Figure 8.1 actually implements the *eventually perfect* FD $\diamond\mathcal{P}$. This FD is weaker than $\mathcal{P}$, since it just assumes that there is some time t after which (SA) must hold whereas $\mathcal{P}$ ensures that (SA) always holds. In our case, t is the time when the last correct process has completed booting and normal mode is entered. Nevertheless, viewed over the whole system lifetime, our FD algorithm only provides eventual semantics:

(SC) *Strong Completeness*: Eventually, every process that crashes is permanently suspected by every correct process.

(E-SA) *Eventually Strong Accuracy*: There is a time after which correct processes are not suspected by any correct process.

As in Section 6.2 we consider f Byzantine processes in order to tolerate early timing faults. (But we may as well refer to the algorithm in Section 6.1.3 for crash faults.) The algorithm given in Figure 8.1 is identical to the algorithm from Section 6.2 (a description can be found there). It just uses the clock synchronization algorithm that handles booting from Section 7.

```
0:  VAR suspect[∀q] : boolean := false;
1:  VAR saw_max[∀q] : integer := 0;

2:  Execute Clock Synchronization from Section 7

3:  if received (round l) from q
4:      → saw_max[q] := max(l, saw_max[q]);
5:  fi

6:  whenever clock value k is updated do (after updating)
7:      → ∀q suspect[q] := (k − Ξ_P) > saw_max[q];
```

Figure 8.1. Failure Detector Implementation

Theorem 26 (Eventually Perfect FD). *Let* $\Xi_{\mathcal{P}} \geq min\{\lceil 3\Omega + 1\rceil, \lceil 2\Omega + 2\rceil\}$. *In a system with* $n \geq 3f + 1$ *processes where booting is considered, the algorithm given in Figure 8.1 implements the eventually perfect failure detector.*

Proof. We have to show that (SC) and (SA) are eventually satisfied. Let t_{up} be the time when the last correct process gets up. Theorem 24 shows that all correct processes satisfy (P), (U), (Q), and (S) by time $t_{up} + 3\tau^+ + \varepsilon$. Hence, after that time, the proof of Theorem 9 applies literally and reveals that our algorithm implements $\mathcal{P}$ and hence belongs to the class $\diamond\mathcal{P}$ during whole system lifetime. □

Theorem 26 reveals that any consensus algorithm that uses $\diamond\mathcal{P}$ under the generalized partially synchronous system model of [19] solves the booting problem if used in

conjunction with our FD implementation. After all, the model of [19] allows arbitrary message losses to occur until the (unknown) global stabilization time GST. $\diamond\mathcal{P}$-based consensus algorithms that work with eventually reliable links can also be used immediately. Such solutions are useful even in the context of real-time systems, since we can bound the time until $\diamond\mathcal{P}$ becomes $\mathcal{P}$. Finally, it should not be too difficult to adapt FD-based consensus algorithms to the perception-based failure model. This would even allow to drop the perfect link assumption after GST.

8.2 General Considerations on Applications during Booting

Due to the eventual semantics of $\diamond\mathcal{P}$ in Section 8.1 it is not required to argue explicitly about how processes might find out C_{max}. This is different in some higher-level applications, e.g. lock-step simulation and atomic broadcast. A simulation algorithm should be able to determine in which round the system is, and when it is safe for a process to start to participate. In case of atomic broadcast the algorithm should not start delivering messages before it can be sure that no earlier messages in between are lost. For this reason we give the following corollary which determines a point in logical time from when on participating in the algorithms is safe. For that reason we add a priori knowledge on the integer clock synchronization precision D_{boot} to the algorithms from now on.

Corollary 3 (Certainty Point). *If a correct process p becomes active at local time ℓ it can be sure that it misses no messages sent by correct processes for rounds $k > \Psi$, where $\Psi = \ell + D_{boot}$. Thus, p has certainty that $p \in \mathcal{P}_{up}(k)$ for all rounds $k > \Psi$. The set of all correct processes that know that they are $\in \mathcal{P}_{up}(k)$ is denoted as $\mathcal{P}_{work}(k)$.*

In the following sections we will see how Ψ can be employed to handle the booting phase for the problems of lock-step simulation, atomic broadcast and NBAC.

8.3 Lock-Step

Lock step algorithms usually assume that all processes start simultaneously in round 0. When system booting is considered this assumption must be weakened since it is impossible in the Byzantine case to guarantee that a process can determine the point in time when all correct processes are up [106]. We hence give a simulation where processes may enter lock-step execution during any round. The result is a simulation for lock-step algorithms with deferred start rounds. In order to keep consistent with the definitions of Section 6.3.3, a process starts a round with sending its message. In Figure 8.2 the boolean *booted* is used to distinguish the start of the first round a processes participates in from other rounds.

In Figure 8.3 the interface to clock synchronization is given. As long as processes remain passive they do not participate (by calling the function *start*) in lock-step algorithms. Upon getting active (`line 5`) the certainty point Ψ is calculated. Based on it the largest missed macro tick m is determined in `line 7`. When the next macro tick $m+1$ is eventually reached (`line 10`), the process starts to participate and sends its

```
0: procedure start(r:integer, booted:boolean)
1: begin
2:     if booted
3:         read round r − 1 messages;
4:         execute round r − 1 computational step;
5:     fi
6:     send round r messages;
7: end;
```

Figure 8.2. Lock-Step Framework using the Synchronizer in Figure 8.3

first message by calling the function *start*. After that it has completed its initialization and participates in the lock-step algorithm (hence just executes `line 16` from then on).

We now show that the given algorithm in fact simulates lock-step behavior. Since we have no simultaneous start in round 0 we just have to make sure that a correct process executes the rounds it participates in correctly.

Lemma 24 (Initialization). *For any correct process p that executes the algorithm given in Figure 8.2 in conjunction with the algorithm in Figure 8.3 with $\Xi_{sync}^{boot} \geq 2\Omega + 1$ there is some r such that p executes all rounds $s \geq r$ in proper order.*

Proof. By Theorem 24 all correct processes become active after sufficiently many correct processes get up. Let correct process p become active at local round k after sufficiently many correct processes are already active. By (P) from now on p increases its round number regularly. By `line 5` and `line 10` in Figure 8.3 process p eventually starts participating. □

In Section 6.3.3 we showed how to choose Ξ_{sync} when sufficiently many processes are always up to guarantee the properties (P), (U), (Q), and (S). (S) was of particular interest there since its time bounds went into calculation of Ξ_{sync}. During booting, however, (S) cannot be guaranteed (since the lower envelope bound does not hold). We will see in the following theorem that (Q) can be employed for this purpose as well.

Theorem 27 (Synchronizer with Booting). *For any correct process p that executes the algorithm given in Figure 8.2 in conjunction with the algorithm in Figure 8.3 with $\Xi_{sync}^{boot} \geq 2\Omega + 1$ it holds that if p executes the computational step from round r it has received all messages sent by correct processes in round r.*

Proof. Let the first process s send its round r message at the corresponding micro tick k. Further let correct process q be the first to experience tick $k+1$ at time t. By (Q) all correct processes $p \in \mathcal{P}_{work}(k)$ experience tick k by time $t + \varepsilon$ and hence send their round r messages which will be received by all correct processes $p \in \mathcal{P}_{work}(k)$ by time $t + \tau^+ + \varepsilon$. By Lemma 3 no correct process can have a greater clock value than $k + 1 + \frac{\tau^+ + \varepsilon}{\tau_f} \leq k + 1 + 2\Omega$ by then. By `line 16` all correct processes have received all round r messages before they execute the computational step for round r. □

```
0:  VAR  r : integer := 0; /* lock step rounds */
1:       m : integer := 0; /* start round */
2:       Ψ : integer := 0 /*certainty point */
3:       participate : boolean := false;

4:  Execute Clock Synchronization from Figure 7.4

5:  upon change to active do
6:     → Ψ := k + D_boot;
7:       m := ⌊Ψ/Ξ^boot_sync⌋;

8:  whenever active and clock k is updated do
9:     → if participate = false
10:        → if k/Ξ^boot_sync ≥ m + 1
11:           → r := ⌊k/Ξ^boot_sync⌋;
12:             call start(r, false);
13:             participate := true;
14:          fi
15:      else
16:        → if k/Ξ^boot_sync ≥ r + 1
17:           → r := r + 1
18:             call start(r, true);
19:      fi
```

Figure 8.3. Synchronizer for the Θ-Model with Startup

Still, upper layer synchronous algorithms that use this synchronizer must be aware that not all processes start simultaneously. There is some work on this problem [69] in the context of consecutive executions of early deciding consensus, where it is required to agree on the starting round for the next instance of consensus.

8.4 Atomic Multicast

The solution for atomic multicast during booting is straight forward. A process starts delivering messages at a given round r only when it can be sure that it can deliver all messages with time stamp $C_s \geq r$ in the same order as delivered by other processes. Any message that originates in a round after certainty point Ψ must have been sent after p got up. Therefore it starts queuing messages from this round on.

Again, we have to adapt the requirements in order to cope with the booting problem. The following starting round requirement prohibits that any process p delivers messages that were possibly sent before p was up.

Starting Round: No process $p \notin \mathcal{P}_{work}(k)$ delivers a message with time stamp k.

The original (UTRB) validity property as stated in Section 6.3.2 requires that all correct processes deliver a message which was broadcast by a correct process. Clearly this cannot be achieved for processes that were down when the message was originally broadcast respectively when it was delivered by the already up processes. The following

booting validity replaces the original one and just requires that the already participating correct processes to deliver messages. Similar argumentation is also required for agreement such that we give an agreement definition in the booting case as well.

Booting Validity: If a correct process broadcasts a message m with time stamp k, then all correct processes $p \in \mathcal{P}_{work}(k)$ eventually deliver m.

Booting Agreement: If a process delivers m with time stamp k, then all correct processes $p \in \mathcal{P}_{work}(k)$ eventually deliver m.

The algorithm is given in Figure 8.4. `lines 1-7` are similar to our atomic broadcast solution without booting in Section 6.3.2. The only difference is delivery of messages in `line 7` which happens only at active processes that have reached a clock value greater than Ψ where they can be sure that messages are ordered consistently. As already seen in the synchronizer implementation, Ψ is calculated at processes upon change to active in `line 11`. Delivery in `line 12` is done similarly to our solution in Section 6.3.2.

```
0:  /* Active Broadcaster s executes if active and C_s > Ψ: */
1:  send (message, s, C_s, i) to all processes
2:  queue (message, s, C_s, i) for delivery

3:  /* Process q ≠ broadcaster executes: */
4:  do upon first receipt of (message, s, C_s, i)
5:      → send (message, s, C_s, i) to all processes
6:        if C_s > Ψ and active
7:          → queue (message, s, C_s, i) for delivery

8:  Execute Clock Synchronization from Figure 7.4

9:  /* Any process p executes: */
10: upon change to active do
11:     → Ψ := k + D_boot;

12: whenever active and clock k is updated do (after updating)
13:     deliver^a all queued messages with time stamp C_s ≤ C_p − Ξ_abc
```

[a]ordered by (1) increasing timestamps, 2) increasing process identifiers and (3) increasing indexes i

Figure 8.4. Atomic Multicast with Startup

Theorem 28 (Atomic Multicast with Booting). *All correct processes $p \in \mathcal{P}_{work}(k)$ that execute the algorithm given in Figure 8.4 deliver all messages from correct processes sent with time stamp $C_s \geq k$ in total order.*

Proof. All processes $p \in \mathcal{P}_{work}(k)$ are up when messages with time stamp $C_s(t) \geq k$ are sent. Since messages are reliably broadcast (in fact multicast since not all processes

need to be up yet) by `lines 1-7` all processes $p \in \mathcal{P}_{work}(k)$ receive these messages consistently.

We may now employ the proof of Theorem 14 (atomic broadcast without booting) since it relies solely on Lemma 3 which also holds during booting since (U) holds. □

8.5 Non-Blocking Atomic Commitment

We now turn to the problem of NBAC and how it can be solved if booting has to be considered. As seen in Section 7.5 clean-crash faults are as severe as crash faults (= asymmetric omission faults) in the booting case. In this section we therefore just consider crash faults—as in Section 6.4.2. As already indicated there, starting all requests in round 0 was just a simplification. In this section each request is issued by some participant which timestamps it. Again, different requests can be separated via the *req* field in the messages. If a correct process reaches the round where it can be sure that it cannot miss any more votes, it decides. In the case of crash faults in Section 6.4.2 we required votes from all n processes. This means that we can only COMMIT requests at times where all processes are active and no processes have crashed yet. From this we derive the following non-triviality property:

Non-Triviality-Booting: For requests issued by some correct process p with time stamp $k = C_p$: If $|\mathcal{P}_{work}(k)| = n$ and all participants vote YES and no failure occurs the outcome decision is COMMIT.

The algorithm given in Figure 8.5 again employs the clock synchronization algorithm from Section 7. Upon changing to active, the certainty point Ψ is set. An active correct process can be sure that it was already active when a request with time stamp greater than Ψ was issued. Therefore, only for requests that have time stamps greater than Ψ the processes actively participate and vote. If for some request all processes participate and no process crashes for Ξ^b_{ac} rounds it is possible to COMMIT if all votes are YES. Note that the *req* field in the algorithm in Figure 8.5 is used to distinguish requests.

Lemma 25. *All participants $p \in \mathcal{P}_{work}(k)$ executing the algorithm given in Figure 8.5 with $\Xi^b_{ac} \geq \Omega(f+2) + D_{boot}$ that decide have delivered the same votes for requests time stamped with clock values $> k$.*

Proof. Since the votes for a request time stamped with k are reliably broadcast by active and participating processes (`line 1-9`), we only have to show that every correct process which delivers a vote does so before it reaches a clock value of $\ell > \Xi^b_{ac} + k$.

At every correct process we have at most two reliable broadcast with causal dependency: Let q be the first correct process which broadcasts its vote at time t. When this message is delivered by any process p, p then reliably broadcasts its own vote. Process p does so by time $t + (f_1 + 1)\tau^+$, f_1 being the faults that happen during this first reliable broadcast. Process p's vote is delivered at any correct process by time $t + (f_1 + 1)\tau^+ + (f_2 + 1)\tau^+$, f_2 being the faults during the second reliable broadcast. Since $f \geq f_1 + f_2$, the latest time any process delivers all

---

To initiate a new request if $k \geq \Psi$
1: send *(k, req, myvote)* to all;
2: deliver *(k, req, myvote)*;

Process $q \neq$ requester executes:
3: if received *(l, req, vote)* for the first time
4: $\rightarrow$ if $l > \Psi$
5: $\rightarrow$ deliver *(l, req, vote)*;
6: send *(l, req, myvote)* to all;
7: deliver *(l, req, myvote)*;

For all processes
8: Execute Clock Synchronization from Section 7

9: upon change to active do
10: $\rightarrow \Psi := k + D_{boot}$;

11: whenever active and clock $k \geq \Psi$ is updated do (after updating)
12: For all requests with time stamp $\leq k - \Xi_{ac}^{b}$
13: **if** all votes are delivered and all are YES
14: $\rightarrow$ return(COMMIT)
15: **else**
16: $\rightarrow$ return(ABORT)
17: **fi**

Figure 8.5. NBAC Protocol Tolerating Crashes with Booting

votes is $t + (f+2)\tau^{+}$. By Lemma 3, by this time the largest possible clock value $\ell \leq \frac{(f+2)\tau^{+}}{\tau_f} + k + D_{boot} = \Omega(f+2) + k + D_{boot} \leq \Xi_{ac}^{b} + k$. □

Theorem 29. *The algorithm given in Figure 8.5 with $\Xi_{ac}^{b} \geq \Omega(f+2) + D_{boot}$ solves the NBAC problem.*

Proof. We show each of the requirements of NBAC separately.
Integrity: A participant decides using a return function, hence only once.

Validity: By `line 15`.

Uniform Agreement: A process commits if it has received all votes, and all votes are YES. By Lemma 25 agreement is achieved.

Termination: By Theorem 24 eventually there will always be progress of clock values in the system. Every correct participating process decides on a request by the time, when it reaches round $\ell > k + \Xi_{ac}^{b}$ by `line 14`. If the request was issued at some time t when progress of clock values was already guaranteed (compare Section 7.5) then very correct process reaches round k by time $t + \tau^{+}\Xi_{ac}^{b}$ by Lemma 5 and decides. If progress of clocks was guaranteed only from some time $t' > t$ on, then all will reach this clock value by time $t' + \tau^{+}\Xi_{ac}^{b}$ by Lemma 5 as well.

Non-Triviality: If no failure occurs every process votes on p's request—which was time stamped with C_p—,and by Lemma 25 every process delivers all votes before it reaches a clock value $k > C_p + \Xi_{ac}^{b}$. It hence commits by `line 15`. □

Chapter 9

Conclusions

9.1 Required Synchrony for Consensus

From a theoretical point of view, when considering new system models one usually starts with examining the solvability of consensus in it. Since the presented clock synchronization algorithms are totally time free (even no a priori knowledge of Θ), they can be employed in several variants of the Θ-Model. We can actually distinguish four possible system models which have assumptions on Θ:

(1) The variant used in this thesis, i.e. some known Θ that always holds. We have provided an implementation of $\mathcal{P}$ in this model. This model is hence well suited for real-time systems.

(2) We can also think of a model where known Θ holds just from some global stabilization time GST on. We have seen that our implementation of $\mathcal{P}$ in model (1) can be employed in order to implement $\diamond\mathcal{P}$ in (2).

(3) Some unknown Θ always holds. In this model $\diamond\mathcal{P}$ can be implemented as well. Whenever a message from a falsely suspected process p drops in, Ξ must be adapted, according to the current local clock value and the round number of p's message. Eventually Ξ will reach the correct value (according to Theorem 9) and our failure detector is perfect from then on.

(4) Some unknown Θ holds from GST on. The same algorithm as in (3) might be employed but in contrast to model (3), Ξ cannot be guaranteed to be a priori bounded according to Theorem 9 since Θ could be violated before GST. Nevertheless we again have an implementation of $\diamond\mathcal{P}$.

The previous arguments just refer to crash faults. When considering more severe types of failures, like Byzantine faults, failure detectors are of no use. For this purpose we provided a simulation which allows to run lock-step algorithms in the Θ-Model in Section 6.3.3. Note also that the consensus algorithms by Dwork, Lynch, and Stockmeyer [37] can easily be transfered into the Θ-Model as well. In [107] we presented a simulation which shows that Byzantine consensus has a solution in a model where the Θ assumptions just holds after some possibly unknown GST.

9.2 Clock Synchronization Implementation

When comparing our novel clock synchronization algorithm (see Section 6.1 and Section 7) to previous ones [106, 109, 67] (which all rely upon the classic non-authenticated clock synchronization algorithm by Srikanth and Toueg [95]) it is obvious that it is much simpler as it requires just one type of messages. Even more important is the property that precision (π) and the upper envelope bound of (α) does not depend on τ^- but rather upon τ_f (the round switching time). As already argued in Section 3.5 this either leads to improved coverage or improved performance (= smaller values for Θ).

This is particularly important in Ethernet-like networks, where two messages cannot be sent over the physical link simultaneously. Since messages are sent strictly one after the other, one may not need to consider the first message for deriving a bound on Θ.

9.3 Dependable Real-Time Systems

Reviewing this thesis's results suggests the following approach to build highly reliable systems: At the basic layer, the clock synchronization with booting (Section 7) should be used. It has the advantage that no a priori knowledge of Θ is required, that it works under a wide variety of process and link failures, and that it delivers the synchrony of the communication layer directly to the upper layer. If the system timing hence temporarily does not behave as assumed, the clock synchronization algorithm keeps working and just gets out of the assumed bounds. If the system returns to the expected behavior the clock synchronization bounds will hold again. This is very important in recently emerging application domains where it is nearly impossible to predict system behavior during the whole system operation. Moreover, for such application we have argued in Section 5.2 that the assumption on the delay ratio is more likely to hold than assumptions on upper bounds on message delays.

Depending on the types of faults, different layers should be put atop of clock synchronization. For crash faults, we propose using the eventually perfect failure detector $\diamond\mathcal{P}$ with booting (Section 8.1). Since it has just eventual semantics, every algorithm that uses $\diamond\mathcal{P}$ will still satisfy its safety properties even when system timing violations translate into false suspicion at the $\diamond\mathcal{P}$ layer. If the system returns to the expected behavior our failure detector becomes perfect again. We can hence guarantee bounded detection times if our assumptions are met and guarantee safety even if our assumptions are violated. This property of asynchronous designs is often ignored in real-time literature [24] which argues that “compare and kill” is the only solution to ensure safety. From the distributed computing theory viewpoint *Doing nothing is safe!* It is of course the task of system engineers to design the interaction with environment in such a way that the previous statement is true. This results in a more robust, safer solution. We believe that many control problems allow such an approach.

On the other hand keeping systems running although time bounds are violated as long as the system remains in a consistent state will usually lead to less severe damage. This is especially true in systems where environment conditions are not as predictable as required for many timed solutions (e.g. space borne applications).

Atop of the $\diamond\mathcal{P}$ layer, finally, higher-level algorithms can be devised. Obviously there lies a drawback in worst case termination times: Algorithms which use eventual type failure detectors are usually less effective than those using perpetual ones since their termination times depend on the failure pattern during the execution as well as on the number and pattern of false suspicions. This must taken into account when dimensioning a reliable distributed system.

There is much recent research [36, 35, 58, 13] on failure detector based asynchronous Byzantine consensus. To our best knowledge, however, there exists no satisfying definition of failure detectors in the case of non-authenticated Byzantine faults by now. We hence propose a different approach to build a system tolerating this kind of faults in the Θ-Model: Applications may directly be put atop our synchronized clocks, using the techniques employed for the solutions of atomic broadcast or non-blocking atomic commitment (cp. Section 8.4 and Section 8.5; these cannot tolerate Byzantine faults, however). Another approach would be to introduce a synchronization layer that creates the illusion of lock-step synchrony (Section 8.3) to the applications. It is then possible to run any synchronous algorithm atop of this layer. This easily allows to devise Byzantine-tolerant applications at the cost of possibly higher termination times.

9.4 Outlook on Further Research Directions

The most challenging question in distributed computing theory is the one on the weakest timing model that allows to solve the fundamental, yet not trivial consensus problem. Therefore it is still subject to currently ongoing work [67, 107, 41, 2, 76]. Although the Θ-Model provides an important step towards this goal, there is still room for improvement. For example, just reconciling some of the mentioned approaches might lead to even weaker timing models.

In the case of the Θ-Model, we mentioned in Section 3.4 that the length ratio of two concurrent message chains is bounded by Θ as consequence of our system model (cp. Section 3.2). Bounding just these length ratio instead of bounding message end-to-end delays might lead to an even weaker system model that allows to solve consensus. One important point, however, must always be regarded: Can the envisioned system model be implemented in real systems? We showed in Section 5.5 that the Θ-Model can in fact be implemented very efficiently in real-time systems. Other approaches [24, 76] that try to weaken the system model, however, just provide probabilistic guarantees. We believe that it is mandatory to show how to implement the new models in order to remain in contact with real systems and real problems.

An issue which is closely related to the timing of distributed systems is queuing. It would be an interesting topic for future research whether we can give analytical feasibility conditions for systems that guarantee applicability of the Θ-Model. As a starting point for such research some networks of queues—as depicted in Figure 2.1—must be analyzed. This analysis, however, must not rely on time triggered processes—as often assumed in queuing theory—but on event based ones and must consider adversaries according to the particular failure model used.

Apart from timing models, there are also open questions on fault models. The self-stabilization [28, 31] paradigm, for example, does not model permanent, but rather arbitrary transient faults. A very interesting research field are fault-tolerant self-stabilizing (ftss) algorithms. Reconciling ftss and time free algorithms leads to even more reliable systems. For the Θ-Model, we have presented ftss failure detector implementations [55]. But many more problems are out there that must be solved.

An interesting research field lies between classic fault-tolerance and ftss. The crash recovery model [3] describes transient faults which do not lead to an arbitrary system state, as assumed in self-stabilization. For the Θ-Model we have solved the booting problem, for example. It seems as if this might be half the way to crash recovery: For example, one might think of a similar non-blocking atomic commit algorithm as the one in Section 8.5 that may be employed in the crash recovery model. We would require that at least $f + 1$ processes do not crash after they have booted. The others may crash and recover. A decision COMMIT can be reached during periods when all processes are active.

Bibliography

[1] Marcos Aguilera, Gérard Le Lann, and Sam Toueg. On the impact of fast failure detectors on real-time fault-tolerant systems. In *Proceedings of the 16th International Symposium on Distributed Computing (DISC'02)*, volume 2508 of *LNCS*, pages 354–369. Springer Verlag, Oct 2002.

[2] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *Proceeding of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*, 2003.

[3] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Failure detection and consensus in the crash-recovery model. *Distributed Computing*, 13(2):99–125, April 2000.

[4] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. On quiescent reliable communication. *SIAM Journal of Computing*, 29(6):2040–2073, April 2000.

[5] Daniel Albeseder. Experimentelle Verifikation von Synchronitätsannahmen für Computernetzwerke. Diplomarbeit, Embedded Computing Systems Group, Technische Universität Wien, May 2004.

[6] Anish Arora, Shlomi Dolev, and Mohamed G. Gouda. Maintaining digital clocks in step. In Sam Toueg, Paul G. Spirakis, and Lefteris M. Kirousis, editors, *Distributed Algorithms, 5th International Workshop*, volume 579, pages 71–79, Delphi, Greece, 7–9 1991. Springer-Verlag.

[7] Hagit Attiya, Danny Dolev, and Joseph Gil. Asynchronous byzantine consensus. *Proceedings of the 3rd ACM Symposium of Distributed Computing*, pages 119–133, August 1984.

[8] Hagit Attiya, Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Bounds on the time to reach agreement in the presence of timing uncertainty. *Journal of the ACM (JACM)*, 41(1):122–152, 1994.

[9] Hagit Attiya and Jennifer Welch. *Distributed Computing*. McGraw-Hill, 1998.

[10] Baruch Awerbuch. Complexity of network synchronization. *Journal of the ACM (JACM)*, 32(4):804–823, 1985.

[11] M.H. Azadmanesh and Roger M. Kieckhafer. Exploiting omissive faults in synchronous approximate agreement. *IEEE Transactions on Computers*, 49(10):1031–1042, October 2000.

[12] Özalp Babaoğlu and Sam Toueg. Non-blocking atomic commitment. In Sape Mullender, editor, *Distributed Systems 2nd Ed.*, pages 147–166. ACM Press, 1993.

[13] Roberto Baldoni, Jean-Michel Hélary, Michel Raynal, and Lenaik Tangui. Consensus in byzantine asynchronous systems. *Journal of Discrete Algorithms*, 1(2):185–210, 2003.

[14] Boaz Barak, Shai Halevi, Amir Herzberg, and Dalit Naor. Clock synchronization with faults and recoveries (extended abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 133–142. ACM Press, 2000.

[15] Marin Bertier, Olivier Marin, and Pierre Sens. Implementation and performance evaluation of an adaptable failure detector. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'02)*, pages 354–363, Washington, DC, June 23–26, 2002.

[16] Martin Biely. An optimal Byzantine agreement algorithm with arbitrary node and link failures. In *Proc. 15th Annual IASTED International Conference on Parallel and Distributed Computing and Systems (PDCS'03)*, pages 146–151, Marina Del Rey, California, USA, November 3–5, 2003.

[17] A. Casimiro, P. Martins, and P. Veríssimo. How to build a timely computing base using real-time linux. In *Proceedings of the 2000 IEEE International Workshop on Factory Communication Systems*, pages 127–134, Porto, Portugal, September 2000. IEEE Industrial Electronics Society.

[18] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *Journal of the ACM*, 43(4):685–722, June 1996.

[19] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[20] Bernadette Charron-Bost. Comparing the atomic commitment and consensus problems. In *Future Directions in Distributed Computing 2002*, volume 2584 of *LNCS*, pages 29–34. Springer-Verlag, 2003.

[21] Bernadette Charron-Bost, Rachid Guerraoui, and André Schiper. Synchronous system and perfect failure detector: Solvability and efficiency issues. In *Proceedings of the IEEE Int. Conf. on Dependable Systems and Networks (DSN'00)*, pages 523–532, New York, USA, 2000. IEEE Computer Society.

[22] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. In *Proceedings IEEE International Conference on Dependable Systems and Networks (ICDSN / FTCS'30)*, 2000.

[23] George Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed Systems*. Assison Wesley, 3rd edition, 2000.

[24] Flaviu Cristian and Christof Fetzer. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):642–657, 1999.

[25] Ariel Daliot, Danny Dolev, and Hanna Parnas. Linear time byzantine self-stabilizing clock synchronization. In *Proceedings of the 7th International Conference on Principles of Distributed Systems*, Dec 2003. to appear.

[26] Ariel Daliot, Danny Dolev, and Hanna Parnas. Self-stabilizing pulse synchronization inspired by biological pacemaker networks. In *Proceedings of the 6th International Symposium on Self-Stabilizing Systems, SSS'03*, volume 2704 of *LNCS*, pages 32–48. Springer Verlag, June 2003.

[27] Susan B. Davidson, Insup Lee, and Victor Wolfe. Timed atomic commitment. In *IEEE Transactions on Computers*, volume 40, pages 573–583, May 1991.

[28] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[29] Danny Dolev, Cynthia Dwork, and Larry Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34(1):77–97, January 1987.

[30] Danny Dolev, Joseph Y. Halpern, and H. Raymond Strong. On the possibility and impossibility of achieving clock synchronization. *Journal of Computer and System Sciences*, 32:230–250, 1986.

[31] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[32] Shlomi Dolev and Jennifer L. Welch. Wait-free clock synchronization. In *Proceeding of the 12th Annual ACM Symposium on Principles of Distributed Computing (PODC'93)*, pages 97–108, 1993.

[33] Shlomi Dolev and Jennifer L. Welch. Self-stabilizing clock synchronization in the presence of byzantine faults. In *Proc. of the 2nd Workshop on Self-Stabilizing Systems*, May 1995.

[34] Shlomi Dolev and Jennifer L. Welch. Wait-free clock synchronization. *Algorithmica*, 18(4):486–511, 1997.

[35] Assia Doudou, Benoit Garbinato, Rachid Guerraoui, and André Schiper. Muteness failure detectors: Specification and implementation. In *Proceedings 3rd European Dependable Computing Conference (EDCC-3)*, volume 1667 of *LNCS 1667*, pages 71–87. Springer, September 1999.

[36] Assia Doudou and André Schiper. Muteness detectors for consensus with byzantine processes. In *Proceedings of the 17th ACM Symposium on Principles of Distributed Computing (PODC-17)*, 1998.

[37] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[38] Cynthia Dwork and Dale Skeen. The inherent cost of nonblocking commitment. In *Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 1–11. ACM Press, 1983.

[39] R. Ernst and W. Ye. Embedded program timing analysis based on path clustering and architecture classification. In *Digest of Technical Papers of IEEE/ACM International Conference on Computer-Aided Design*, pages 598–604. IEEE Computer Society, April 1997.

[40] Christof Fetzer, Michel Raynal, and Frederic Tronel. An adaptive failure detection protocol. In *Pacific Rim International Symposium on Dependable Computing (PRDC 2001)*, December 2001.

[41] Christof Fetzer and Ulrich Schmid. On the possibility of consensus in asynchronous systems with finite average response times. Research Report 14/2004, Technische Universität Wien, Institut für Technische Informatik, Treitlstraße 3, A-1040 Vienna, Austria, 2004. (submitted).

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

[43] Eli Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 143–152. ACM Press, 1998.

[44] Vijay K. Garg and J. Roger Mitchell. Implementable failure detectors in asynchronous systems. In *Proceedings of the 18th Int. Conference on Foundations of Software Technology and Theoretical Computer Science (FST & TCS'98)*, LNCS 1530, pages 158–169. Springer, 1998.

[45] Jim N. Gray. Notes on data base operating systems. In G. Seegmüller R. Bayer, R.M. Graham, editor, *Operating Systems: An Advanced Course*, volume 60 of *Lecture Notes in Computer Science*, chapter 3.F, page 465. Springer, New York, 1978.

[46] R. Guerraoui and A. Schiper. The decentralized non-blocking atomic commitment protocol. In *Proceedings of the 7th IEEE Symposium on Parallel and Distributed Processing (SPDP-7)*, pages 2–9, San Antonio, Texas, USA, 1995.

[47] Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15:17–25, 2002.

[48] Indranil Gupta, Tushar D. Chandra, and Germán S. Goldszmidt. On scalable and efficient distributed failure detectors. In *Proceedings of the 20th ACM Symposium on Principles of Distributed Computing (PODC'01)*, pages 170–179, August 2001.

[49] J.C. Palencia Gutiérrez, J.J. Gutiérrez Garcia, and M. Gonzáles Harbour. Best-case analysis for improving the worst-case schedulability test for distributed hard real-time systems. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, pages 35–44, June 1998.

[50] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In Sape Mullender, editor, *Distributed Systems*, chapter 5, pages 97–145. Addison-Wesley, 2nd edition, 1993.

[51] J.-F. Hermant and Gérard Le Lann. Fast asynchronous uniform consensus in real-time distributed systems. *IEEE Transactions on Computers*, 51(8):931–944, August 2002.

[52] J.-F. Hermant and Josef Widder. Implementing time free designs for distributed real-time systems (a case study). Research Report 23/2004, Technische Universität Wien, Institut für Technische Informatik, May 2004. Joint Research Report with INRIA Rocquencourt. (submitted for publication).

[53] M. Hurfin and M. Raynal. Asynchronous protocols to meet real-time constraints: Is it really feasible? How to proceed? In *Proceedings of the IEEE Intl. Symposium on Object-Oriented Real-Time Distributed Computing (ISORC)*, pages 290–297. IEEE Computer Society, April 2003.

[54] Martin Hutle. An efficient failure detector for sparsely connected networks. In *Proc. IASTED International Conference on Parallel and Distributed Computing and Networks (PDCN'04)*, Innsbruck, Austria, February 2004.

[55] Martin Hutle and Josef Widder. Time free self-stabilizing local failure detection, May 2004. (submitted for publication).

[56] E.D. Jensen and B Ravindran. Guest editors' introduction to special section on asynchronous real-time distributed systems. *IEEE Transactions on Computers*, 51(8):881–882, August 2002.

[57] Roger M. Kieckhafer, Chris J. Walter, Alan M. Finn, and Philip M. Thambidurai. The MAFT architecture for distributed fault tolerance. *IEEE Transactions on Computers*, 37:398–405, April 1988.

[58] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1):16–35, 2003.

[59] Hermann Kopetz. *Real-Time Systems. Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[60] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[61] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Efficient algorithms to implement unreliable failure detectors in partially synchronous systems. In *Proceedings of the 13th International Symposium on Distributed Computing (DISC'99)*, LNCS 1693, pages 34–48. Springer, September 1999.

[62] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Optimal implementation of the weakest failure detector for sloving consensus. In *Proceedings of the 19th ACM Symposium on Principles of Distributed Computing (PODC'00)*, page 334, 2000.

[63] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. On the impossibility of implementing perpetual failure detectors in partially synchronous systems. In *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing (PDP'02)*, January 2002.

[64] Gérard Le Lann. Certifiable critical complex computing systems. In K. Duncan and K. Krueger, editors, *Proceedings 13th IFIP World Computer Congress 94*, volume 3, pages 287–294. Elsevier Science B.V. (North-Holland), 1994.

[65] Gérard Le Lann. On real-time and non real-time distributed computing. In *Proceedings 9th International Workshop on Distributed Algorithms (WDAG'95)*, volume 972 of *Lecture Notes in Computer Science*, pages 51–70. Springer, September 1995.

[66] Gérard Le Lann. Asynchrony and real-time dependable computing. In *Proceedings of the 8th IEEE Intl. Workshop on Object-Oriented Real-Time Dependable Systems (WORDS 2003)*, page 8p., Guadalajara, Mexico, 2003. IEEE Computer Society.

[67] Gérard Le Lann and Ulrich Schmid. How to implement a timer-free perfect failure detector in partially synchronous systems. Technical Report 183/1-127, Department of Automation, Technische Universität Wien, January 2003. (submitted).

[68] Gérard Le Lann and Ulrich Schmid. How to maximize computing systems coverage. Technical Report 183/1-128, Department of Automation, Technische Universität Wien, April 2003.

[69] Yehuda Lindell, Anna Lysyanskaya, and Tal Rabin. Sequential composition of protocols without simultaneous termination. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 203–212. ACM Press, 2002.

[70] Jennifer Lundelius-Welch and Nancy A. Lynch. A new fault-tolerant algorithm for clock synchronization. *Information and Computation*, 77(1):1–36, 1988.

[71] Nancy Lynch. *Distributed Algorithms*. Morgan Kaufman, 1996.

[72] Stephen R. Mahaney and Fred B. Schneider. Inexact agreement: Accuracy, precision, and graceful degradation. In *Proceedings 4th ACM Symposium on Principles of Distributed Computing*, pages 237–249, Minaki, Canada, August 1985.

[73] Paul S. Miner. Verification of fault-tolerant clock synchronization systems. *NASA Technical Paper 3349*, November 1993.

[74] Shivakant Mishra and Cristof Fetzer. The timewheel group communication system. *IEEE Transactions on Computers*, 51(8):883–899, August 2002.

[75] Achour Mostéfaoui, David Powell, and Michel Raynal. A hybrid approach for building eventually accurate failure detectors. In *Proceedings of the 10th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2004), 3-5 March 2004, Papeete, Tahiti*, pages 57–65. IEEE Computer Society, 2004.

[76] Anchour Mostefaoui, Eric Mourgaya, and Michel Raynal. Asynchronous implementation of failure detectors. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*, San Francisco, CA, June 22–25, 2003.

[77] Sape Mullender. *Distributed Systems*. ACM Press/Addison Wesley, New York, 2nd ed. edition, 1993.

[78] Marina Papatriantafilou and Philippas Tsigas. Self-stabilizing wait-free clock synchronization. Technical Report CS-R9421, Centrum voor Wiskunde and Informatica, Netherlands, 1994.

[79] Marina Papatriantafilou and Philippas Tsigas. On self-stabilizing wait-free clock synchronization. *Parallel Processing Letters*, 7(3):321–328, 1997.

[80] S. Ponzio. The real-time cost of timing uncertainty: Consensus and failure detection. Master's thesis, Massachusetts Institute of Technology, May 1991.

[81] Stephen Ponzio and Ray Strong. Semisynchrony and real time. In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92)*, pages 120–135, November 1992.

[82] Parameswaran Ramanathan, Kang G. Shin, and Ricky W. Butler. Fault-tolerant clock synchronization in distributed systems. *IEEE Computer*, 23(10):33–42, October 1990.

[83] Michel Raynal. A case study of agreement problems in distributed systems: non-blocking atomic commitment. In *Proceedings of High-Assurance Systems Engineering Workshop*, pages 209–214, Aug 1997.

[84] Ulrich Schmid, editor. *Special Issue on The Challenge of Global Time in Large-Scale Distributed Real-Time Systems*, Real-Time Systems 12(1–3), 1997.

[85] Ulrich Schmid. Orthogonal accuracy clock synchronization. *Chicago Journal of Theoretical Computer Science*, 2000(3):3–77, 2000.

[86] Ulrich Schmid. How to model link failures: A perception-based fault model. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'01)*, pages 57–66, Göteborg, Sweden, July 1–4, 2001.

[87] Ulrich Schmid and Christof Fetzer. Randomized asynchronous consensus with imperfect communications. Technical Report 183/1-120, Department of Automation, Technische Universität Wien, January 2002. (Extended version of [88]).

[88] Ulrich Schmid and Christof Fetzer. Randomized asynchronous consensus with imperfect communications. In *22nd Symposium on Reliable Distributed Systems (SRDS'03)*, pages 361–370, October 6–8, 2003.

[89] Ulrich Schmid and Klaus Schossmaier. Interval-based clock synchronization. *Real-Time Systems*, 12(2):173–228, March 1997.

[90] Ulrich Schmid and Bettina Weiss. Synchronous Byzantine agreement under hybrid process and link failures. Technical Report 183/1-124, Department of Automation, Technische Universität Wien, November 2002. (replaces TR 183/1-110).

[91] Ulrich Schmid, Bettina Weiss, and John Rushby. Formally verified byzantine agreement in presence of link faults. In *22nd International Conference on Distributed Computing Systems (ICDCS'02)*, pages 608–616, July 2-5, 2002.

[92] Fred B. Schneider. Understanding protocols for byzantine clock synchronization. Technical Report 87-859, Cornell University, Department of Computer Science, August 1987.

[93] Barbara Simons, Jennifer Lundelius-Welch, and Nancy Lynch. An overview of clock synchronization. In Barbara Simons and A. Spector, editors, *Fault-Tolerant Distributed Computing*, pages 84–96. Springer Verlag, 1990. (Lecture Notes on Computer Science 448).

[94] Paul Spirakis and Basil Tampakas. Efficient distributed algorithms by using the archimedean time assumption. In *Proceedings of the 5th Annual Symposium on Theoretical Aspects of Computer Science (STACS 88)*, volume 294 of *LNCS*, pages 248–263. Springer Verlag, February 1988.

[95] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[96] Wilfried Steiner and Michael Paulitsch. The transition from asynchronous to synchronous system operation: An approach for distributed fault-tolerant systems. *Proceedings of the The 22nd International Conference on Distributed Computing Systems*, July 2002.

[97] Andrew S. Tanenbaum and Maarten van Steen. *Distributed Systems. Principles and Paradigms.* Prentice Hall, 2001.

[98] Gerard Tel. *Introduction to Distributed Algorithms.* Cambridge University Press, 2001.

[99] Paulo Veríssimo and António Casimiro. The timely computing base model and architecture. *IEEE Transactions on Computers*, 51(8):916–930, August 2002.

[100] Paulo Veríssimo, António Casimiro, and Christof Fetzer. The timely computing base: Timely actions in the presence of uncertain timeliness. In *Proceedings IEEE International Conference on Dependable Systems and Networks (DSN'01 / FTCS'30)*, pages 533–542, 2000.

[101] Paulo Veríssimo and Luís Rodrigues. *Distributed Systems for System Architects.* Kluwer Academic Publishers, 2001.

[102] Paul M.B. Vitányi. Distributed elections in an archimedean ring of processors. In *Proceedings of the sixteenth annual ACM symposium on Theory of computing*, pages 542–547. ACM Press, 1984.

[103] Paul M.B. Vitányi. Time-driven algorithms for distributed control. Report CS-R8510, C.W.I., May 1985.

[104] Chris J. Walter and Neeraj Suri. The customizable fault/error model for dependable distributed systems. *Theoretical Computer Science*, 290:1223–1251, October 2002.

[105] Bettina Weiss and Ulrich Schmid. Consensus with written messages under link faults. In *20th Symposium on Reliable Distributed Systems (SRDS'01)*, pages 194–197, October 28–31, 2001.

[106] Josef Widder. Booting clock synchronization in partially synchronous systems. In *Proceedings of the 17th International Symposium on Distributed Computing (DISC'03)*, volume 2848 of *LNCS*, pages 121–135. Springer Verlag, October 2003.

[107] Josef Widder. Consensus in the presence of bounded asynchrony: A simulation (extended abstract), May 2004. (submitted for publication).

[108] Josef Widder, Gérard Le Lann, and Ulrich Schmid. Perfect failure detection with booting in partially synchronous systems. Technical Report 183/1-131, Department of Automation, Technische Universität Wien, April 2003.

[109] Josef Widder and Ulrich Schmid. Booting clock synchronization in partially synchronous systems with hybrid node and link failures. Technical Report 183/1-126, Department of Automation, Technische Universität Wien, January 2003. (submitted for publication).

Curriculum Vitae

Josef Widder
1110 Vienna, Meidlgasse 41/4/4

Personal Data

Date of Birth:	March, 18, 1977
Place of Birth:	Vienna
Citizenship:	Austria

Education

1983 – 1987	Volksschule *(elementary school)* 1110 Wien, Hoefftgasse 7
1987 – 1991	Bundesrealgymnasium *(grammar school)* 2320 Schwechat, Ehrenbrunngasse 6
1991 – 1996	Höhere Technische Bundeslehranstalt Abteilung für EDV/Betriebstechnik *(polytechnic - EDP/engineering department)* 1030 Wien, Ungargasse 69
1996 – 2002	Technische Universität Wien – Informatik *(Vienna University of Technology – Computer Science)* academic degree: Diplomingenieur *(Master of Science)*

Working Experience

July 1993 and July 1995	summer job at Fa. Zörkler gearwheel technologies & mechanical engineering
July – August 1997	summer job at Spardat
Sept 1997 – Jan 1998	working student at Spardat: Software developer
September 2000	Die Donau - Danube Tourist Commission Homepage development *danube-river.org* (*Special Award of the Jury* was given by the magazine "tourist austria" in June 2001)
May 2001–May 2002	volunteer: Aktion Leben Österreich development of donator/donations database
Jan 2003 – present	research assistant at TU Vienna Automation Systems Group and Embedded Computing Systems Group