DISSERTATION

Failure Detection in Sparse Networks

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

Univ.Prof. Dr.techn. Ulrich Schmid

Inst.-Nr. E182/2
Institut für technische Informatik
Embedded Computing Systems Group

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

Dipl.Ing. Martin Hutle

Matr.-Nr. 9626133

Leystraße 110/1/14
1200 Wien

Wien, im August 2005 _____

Kurzfassung

Einer der meist erforschten Ansätze um die Unlösbarkeit des Consensus Problems in vollständig asynchronen Systemen zu umgehen, ist das 1996 von Chandra und Toueg in einem wegweisenden Artikel vorgestellte Konzept der Fehlerdetektoren. Seither haben sich Fehlerdetektoren nicht nur als theoretische Abstraktion der notwendigen Synchronität für Consensus bewährt, sondern sie bilden auch nützliche Bausteine für viele Algorithmen im Bereich der Fehlertoleranten Verteilten Systeme.

Daher findet sich auch in der Literatur eine Menge an Ansätzen, Fehlerdetektoren möglichst effizient zu implementieren. Allerdings basieren praktisch alle bisherigen Lösungen auf der Annahme eines vollverbundenen Netzwerks zwischen den einzelnen Prozessoren des Systems.

Diese Arbeit beschäftigt sich mit der Implementierung verschiedener Fehlerdetektoren in Sparse Networks. Unvollständige Graphen modellieren die Gegebenheiten von vielen Low-Level Netzwerken, wie etwa von Wireless Ad-Hoc Networks, Sensor Networks und auch des Internets. Dieser Ansatz erweist sich als sehr nützlich, da—wie diese Arbeit zeigt—Algorithmen nicht nur effizienter werden, sondern auch direkt von dieser Netzwerkstruktur profitieren können. Andererseits sind Beweise in solchen Graphen meist aufwändiger und unübersichtlicher als für vollverbundene Netze. Durch die Einführung von lokalen Fehlerdetektoren und entsprechenden Transformationsalgorithmen zu globalen Fehlerdetektoren können die Beweise einfacher und eleganter werden.

Diese Arbeit beleuchtet verschiedene Aspekte von Fehlerdetektorimplementierungen in Sparse Networks: Konstante Nachrichtengröße trotz beliebig großer Netze, geringe Synchronitätsannahmen und die Kombination mit Selbststabilisierung. Gerade letzteres scheint eine sehr interessante Zusammenführung zweier verschiedener Ansätze der Fehlertoleranz zu sein, die bisher noch nicht ausreichend betrachtet wurde.

Abstract

One of the most explored approaches to overcome the impossibility of distributed consensus in fully asynchronous systems was the introduction of the concept of unreliable failure detectors by Chandra and Toueg in 1996. Failure detectors emerged not only as theoretical encapsulation of the synchrony needed for consensus, but also as useful building blocks for many distributed algorithms.

Therefore, in literature a lot of effort has been spent to implement such failure detectors efficiently. However, almost all currently known solutions are based on the assumption of a fully connected network between the processors of the system.

This thesis focuses on the implementation of various failure detectors in sparsely connected networks. Sparse networks model the nature of many non-broadcast networks on a low-level basis, namely wireless ad-hoc networks, sensor networks, and even the Internet. From such an approach, an implementation can profit regarding message complexity and accuracy. However, proving the correctness of a failure detector algorithm in sparse networks showed up to be an elaborating task. The introduction of a local failure detector as a new class of failure detectors with appropriate transformation algorithms to global failure detectors simplifies these things.

The failure detector implementations in this work cover several system models that head at distinct goals, including constant message size in arbitrary large networks, weak timing, and self-stabilization. Especially the latter seems to be a very interesting combination of two flavors of fault tolerance—robustness and self-stabilization—which has not been addressed sufficiently in literature.

Acknowledgment

I am grateful to my advisor, Prof. Ulrich Schmid. He inspired my interest in distributed computing and supported me in many ways during the becoming of this thesis. I thank my second assessor, Prof. Felix Freiling. He helped me a lot with his detailed and precious comments.

Many thanks go to Josef Widder. The discussions with him were always enlightening and I enjoyed the joint work with him a lot. Parts of this thesis originates also from this joint work. I like to thank Martin Biely and Bettina Weiss for proofreading and comments on parts of this thesis.

I am grateful to my family and to my friends, who provided the right environment for my studies and my work. Thanks go also to all other people that contributed in any way to the success of this thesis.

Contents

Chapter 1

Introduction

A *distributed system* comprises a set of autonomous computing entities, connected by a network, that perform a common task. Such a system is *fault-tolerant*, if this common task is also achieved even if parts of the system fail.

In our abstraction, we use the term *process* for a computational entity. In literature, sometimes processors are considered instead of processes. Since such an approach makes only a difference if we consider processes *and* processors (meaning, if a processor crashes, also all processes located at this processor crash), we use processes as computational entities for our model.

The network is also considered as an abstract device, via which processes can communicate by sending messages to other processes. Most work in the context of fault-tolerant distributed systems considers *fully connected networks* fully connected, i.e., networks, where every process can communicate with every other process in the system in a direct way. In real systems, this is no natural property: processing entities are physically connected only with a small number of neighbors. To ensure point-to-point communication, networks layers (e.g. routing, packet assembly and disassembly, load balancing, etc.) have to be implemented. As we will see, using such an approach for our problems has serious drawbacks. Thus, this thesis is devoted to the implementation of algorithms *directly* on the underlying sparse network.

1.1 The Fault-tolerant Consensus Problem

The consensus problem is a fundamental problem in fault-tolerant distributed computing. Intuitively speaking, it requires the processes of the system to agree on a common decision value.

The consensus problem is defined as follows: We consider a network of processes where some of these processes may fail by crashing. Every process has a single input value from a fixed alphabet, and at the end of the execution it outputs a single output value from

the same alphabet, fulfilling the following properties:

Termination In every execution, every nonfaulty process eventually decides.

Agreement In every execution, no two nonfaulty processes decide on different values.

Validity In every execution, if all the processes have the same input, then any value decided upon must be that common input.

The simplest form of consensus, *binary consensus* considers only the values 0 and 1. Note that there is also a stronger variant of consensus called *uniform consensus*, that prohibits also faulty processes to decide on conflicting values. There exist many solutions for the consensus problem in literature. However, due to the impossibility shown by Fischer, Lynch and Patterson in 1985, none of these solution works in a purely asynchronous system if there is only a single process that may crash:

Theorem 1.1. *[FLP85] No [deterministic] consensus protocol is totally correct in spite of one fault [in an asynchronous system]*

This impossibility stems from the following fact: In a purely asynchronous system there is no bound on the transmission delay, i.e., a message may be arbitrarily long in transit from a process p to another process q. On the other hand, processes may fail by crashing. Now, if p waits for a message from q, it can be never sure, whether this process has crashed or if just the communication is slow. Whatever p does may be the wrong decision: If it waits without bound it may wait forever, since q may have crashed—violating the termination property. If it stops waiting for the message and continues without this message, the message might have been just slow, and p decides without the information from q, which may lead to a violation either of the agreement or the validity property.

Due to the FLP impossibility, a lot of research [DDS87, DLS88, PS92, ADLS94] has been made for answering the question: How much synchrony is needed for solving consensus? In [DDS87], this ended up in 32 (!) different system models constructed from 5 binary assumptions regarding synchrony:

- asynchronous/synchronous processes
- asynchronous/synchronous communication
- asynchronous/synchronous message order
- point-to-point/broadcast transmission
- atomic/separate receive and send.

For each combination of these, an analysis whether consensus is solvable or not was made.

One of the most popular approaches for circumventing the FLP impossibility is the concept of *partial synchrony* [DLS88]. It extends the FLP model by adding an absolute upper bound on message transmission delays (not end-to-end) and an upper bound on the relative computational speeds of any two processes

These bounds need not necessarily be known a priori to the processes: In [DLS88], these bounds can either be unknown, or known but hold only after some unknown global stabilization time GST. A generalized partially synchronous model integrating those two models has been introduced in [CT96]: It assumes that relative speeds, delays and message losses are arbitrary up to GST; after GST, no message may be lost and all relative speeds and all communication delays must be smaller than the (possibly) unknown upper bounds. This model allows to solve consensus while the synchrony assumption has a high assumption coverage. In fact, the model is violated only if transmission times are increasing forever, which is very unlikely in real systems. Partial synchrony is addressed in more detail in Section 2.5.3.

Note that the FLP impossibility holds only for deterministic algorithms: With the help of *randomization*, it is possible to implement an algorithm that reaches agreement and terminates with probability 1 [BO83, Völ04] in a crash prone asynchronous system. Note that when using such an approach the agreement property is still guaranteed and not associated with a probability.

1.2 Failure Detectors

In their seminal paper [CT96], Chandra and Toueg presented a way to solve consensus in an asynchronous system by introducing the concept of a *failure detector* . Intuitively speaking, a failure detector is a module located at every process that gives this process a "hint", whether another process may have crashed or not. To solve consensus, this information needs not to be reliable, i.e., the failure detector is allowed to make mistakes. Thus such a failure detector is called an *unreliable failure detector*.

However, the use of a failure detector does not invalidate the FLP impossibility. Thus, every failure detector implementation requires some synchrony from the system it is implemented in. Nevertheless, the algorithm that uses the failure detector can be written totally asynchronous. Therefore, a failure detector can be seen as an encapsulation of the synchrony of the system.

Later research showed that bounds on message transmission delays are not the only way to implement a failure detector [BKM97]. However, for an application it is of no concern, how the failure detector is implemented. This is one of the main advantages of the use of failure detectors: They provide an abstraction of the required synchrony and allow application algorithms to forget about time. Further, if there are several processes on a processor and the major failure source is a crash of the whole processor, a single failure detector module may be used by all processes located at at that processor.

Encapsulating synchrony in failure detectors has also a theoretical advantage: It allows to compare problems regarding the required synchrony (cf. e.g. the weakest failure detector for consensus, Section 3.4,[CHT96]).

1.3 Sparse Networks

As indicated above, all existing implementations of failure detectors rely on a fully connected network. Although this is a convenient assumption when writing the failure detector implementation, this is not a very attractive choice:

- Most real networks are *not* fully connected a priori: Processes are physically capable to communicate only with a certain subset of other processes, their neighbors. Wireless ad-hoc networks, sensor networks and also the Internet are good examples for systems with a non-fully connected structure of the communication network. But even in distributed systems that use a shared broadcast medium like a bus, the whole network is often composed of subnetworks via gateways, thus also resulting in a non-complete topology of the communication graph.

- In such networks, point-to-point communication is implemented using routing algorithms. First of all, if routing is not needed except for the failure detector this is an overkill, especially if the network is composed of a lot of tiny nodes, like in a sensor network. If routing is needed anyway, in a fault-prone network the routing algorithm needs information if other nodes in the system may have crashed—exactly the information a failure detector provides. But the routing algorithm can not use the failure detector if the correctness of the failure detector depends on the routing itself. At least we get a circular relationship of routing and failure detection that is difficult to prove correct and violates the principle of modularity.

- Finally, as we have seen above, at least some knowledge about the transmission delays of messages is vital for failure detection. Most of the uncertainty about these delays is not due to the physical transmission time but due to transmission buffers and scheduling issues at the processes. Using a routing protocol increases the uncertainty about timing: additional buffer and processing steps at several processes are inserted, retransmissions and heuristics make the transmission time almost unpredictable.

On the other hand, if failure detectors are located close to the low-level network and failure detector messages are equipped with sufficient high priorities, communication takes place only with physical neighbors and retransmissions of lost messages are not necessary, most of the effects denoted above can be canceled. Since timeouts are smaller in such a system the failure detector also provides a faster service and thus application algorithms will terminate faster. This fits also to the fast failure detector approach [ALLT02].

1.4 Self-stabilization

Until now we considered only one kind of fault-tolerance, which is also called *robustness* [Tel94]: Faults are bounded in the space dimension but may be unbounded in time. A common assumption is e.g. that f out of n processes may permanently fail. *Self-stabilization* is fault-tolerance against faults that are bounded in time but unbounded otherwise: Up to some unknown time the system may behave arbitrarily and end up in an arbitrary state. After that, the system behaves normally forever and the algorithm has to converge from such a possibly erroneous state to a legitimate state in finite time and remain in a legitimate state forever. No further failures are allowed after the stabilization time.

Self-stabilization was introduced by Dijkstra [Dij74] and has since then become well studied topic of distributed computing—as has robustness. The combination of both types of fault tolerance, however, does not appear often in literature. This is all the more surprising, since there are serious reasons to consider this approach:

- Self-stabilization increases the fault-tolerance of robust algorithms: Consider a situation where the failure model for the robust algorithm is violated. A non-stabilizing algorithm might get stuck in some incorrect state forever. With self-stabilization, the service can recover automatically after the system assumptions continue being true.

- Robustness increases the fault-tolerance of self-stabilizing algorithms: If a process fails permanently, self-stabilization cannot recover, since the system will never stabilize.

- For long living applications, reintegration of formerly crashed processes is necessary to always keep a sufficient number of correct processes in the system. Although there are approaches in the robustness domain [ACT00], they are very complex. If self-stabilization is employed, processes can simply be added to the system and will be automatically integrated.

In fact, there are only a few publications that consider the combination of failure detection and self-stabilization and many important topics are still not addressed in this field.

1.5 Message-driven Algorithms

Generally, the discipline of distributed computing considers sets of distributed processes that execute algorithms where each execution consists of a sequence of events. In the context of reliable agreement problems, much work [CT96, DDS87, DLS88] focuses on timing constraints of these events; e.g. upper bounds between send and reception events of messages between processes (see Section 2.5). Another issue is event generation. We

distinguish here two kinds of models, i.e., time-driven and message-driven. In time-driven algorithms, events occur due to passage of time and are triggered by clocks or timers. In contrast, when considering message-driven algorithms, after the algorithm was started, all events happen as immediate reaction to a received message while clocks are either not part of the model or are just not employed by the algorithms.

Note, however, that the issues of timing constraints and event generation are orthogonal. Consider, e.g., the well known failure detector based consensus algorithms of [CT96] which work in an asynchronous model of computation (often referred to as "time-free" model, reflecting the absence of timing bounds). These algorithms must be attributed as time-driven as steps can be taken—independently of the presence or absence of messages in input buffers—just by the passage of time respectively the progress of the program counter. It seems obvious that solutions to the same problem can be achieved with message-driven algorithms if

1. messages are immediately processed upon reception,

2. the failure detector module triggers the consensus algorithm if new suspicions have been added and

3. the failure detector implementation itself is message-driven.

Most existing failure detector implementation in the literature [CT96, ADGFT03a, BKM97] are not message-driven as they periodically send messages (e.g. heartbeats). Exceptions are the message-driven failure detector implementations of [LLS03, WLLS05] which show that failure detectors can be implemented without autonomous event generation (i.e., timers or clocks).

A self-stabilizing algorithm cannot be purely message-driven: if all messages get lost during the instable period, the algorithm is not able to make any progress and thus cannot perform any useful task. However, if we enhance such a system with a weak module that unlocks the algorithm from time to time, failure detector implementations are possible. We discuss the topic of such *deadlock prevention events* in Section 7.2.

1.6 Related Work

The topic of implementing a failure detector on a sparse network is relatively unexplored. In the self-stabilization literature, local failure detectors have been introduced [BKM97] to solve classical self-stabilizing problems in sparse networks with crashing processes, which would otherwise remain unsolvable [AH93]. Closely related to sparse networks is the problem of partitionable environments. Most literature on this topic comes from work on the *group membership problem* [BDM01]. Failure detection in partitionable systems is addressed in [FKM+95] and [ACT99]. Systems with omission failures are investi-

gated in [DFKM97] and more recently in [DGFF05]. Recovering processes are considered in [ACT00], but are not in the context of this thesis.

Another aspect that has been considered is the question how more severe faults, like byzantine process behavior, can be integrated into the failure detector approach [DS98, DGG02, DGGS99, KMMS03]. Although this is a very demanding topic and not sufficiently solved in the literature it is out of the scope of this thesis.

The whole work on sparse network failure detector implementation blends nicely with the *fast failure detector* approach of [ALLT02]. Implementing failure detectors directly on the sparse network and not using point-to-point application messages allows the system to implement a fast and timely service for failure detector messages nearly independent of the load produced by application messages. By doing so, we get not only a gain in assumption coverage of the timing model for failure detector messages, but also applications that use a fast failure detector terminate earlier.

1.7 Motivation

In this thesis, we show how the implementation of a failure detector can profit from a sparse network model. We do so by heading at distinct goals: We show that such a sparse network graph model,

- does not require higher assumption on synchrony.
- can be used to reduce the message complexity of algorithms.
- does not impair the achievable fault-tolerance properties. If combined with the self-stabilization paradigm, fault-tolerance can be even improved.

1.8 Structure of this Thesis

In Chapter 2, a detailed system model for all following chapters is given. Since the system assumptions in these chapters differ, the overall system model is chosen such that it covers all these system model. Chapter 3 gives an overview of existing failure detector specifications and defines the new class of local failure detectors. Synchrony is considered in Chapter 4. We adapt an algorithm from Aguilera et al. [ADGFT03b] to work in a sparse network. The original algorithm works in the weakest model we know of where consensus can be solved. Thus we show that when choosing a sparse network as communication system, no additional synchrony assumptions are needed. Moreover, we even get some gain in efficiency and in accuracy. On the other hand, the resilience is not reduced. In Chapter 5 we focus on message efficiency: We show how it is possible to profit from

the sparse topology. Self-stabilization is introduced in Chapter 6. In Chapter 7 we discuss the conditions for message-driven self-stabilizing algorithms and provide appropriate algorithms. The thesis concludes with a summary of results in Chapter 8.

Chapter 2

System Model

In this chapter we describe a general model for a timed distributed message-passing system based on a network with arbitrary topology.

This model is chosen such that it covers all cases in the following chapters. The models used in the these chapters are special cases of the global model. For instance, in the Chapters 4 and 5 no self-stabilization is needed whereas in Chapter 7 we use the full self-stabilization properties of the system model.

2.1 Time and Clocks

We assume the existence of a discrete global clock with values from a set $\mathcal{T} = \mathbb{R}$, which is used only for analysis and is not available to the processes.

Some algorithms, however, require a local clock, such that processes can measure time intervals. For simplicity we assume perfect clocks, but our results can be easily adapted to systems with bounded drift clocks. Algorithms that need such a device are called *time-driven*, else they are *time-free*. An algorithm that does not use any clock or clock-like device can only react to the message pattern it perceives. Thus such algorithms are *message-driven*.

We are aware of the possibility of counting steps to derive some sort of clock-like device. Although such an implementation is not bound to a specific hardware device, we consider this just as another form of a clock implementation, and classify therefore an algorithm that uses such a technique as time-driven. Further, as our results in Chapter 7 show, a local clock implementation obtained by sending messages to itself is not equivalent to a real local clock: in the self-stabilizing case, these messages can get lost or may be corrupted and thus do not guarantee clock progress.

2.2 Network Model

Our system comprises a set $\Pi = \{1 \dots n\}$ of processes, where each process is a state machine. Processes communicate by *message-passing* over links. An edge $\lambda = (p, q) \in \Lambda$ of the communication graph $G = (\Pi, \Lambda)$, stands for an unidirectional link from p to q. We say a graph is *fully connected* if $\Lambda = \Pi \times \Pi$, otherwise it is *sparse*. Generally, processes and links may fail by crashing, i.e., they stop executing their state machine permanently. In order to capture this fact, we consider the communication graph to be time dependent, i.e., the graph $G(t) = (\Pi, \Lambda(t))$ changes with time $t \in \mathcal{T}$. $\Lambda(t)$ contains an edge $\lambda = (p, q)$ if and only if there is a direct link from p to q which has not crashed by t. For some cases we will restrict our model to process crashes resp. link crashes only. We say a link (p, q) is *bidirectional* if $(p, q) \in \Lambda(t) \Rightarrow (q, p) \in \Lambda(t)$ holds.

For an arbitrary but fixed time t, the *distance* $D(p, q, t)$ of two nodes p and q denotes the length of the shortest path from p to q in $G(t)$. If no such path exists, we say the nodes are not connected and write $D(p, q, t) = \infty$. The longest distance in the network is called the *diameter* $d(t) = \max_{p,q \in \Pi} D(p, q, t)$. Two nodes connected directly by a link are called *neighbors*, the set of all neighbors of a process p at time t is denoted by $\mathrm{nb}(p, t) = \{q | (q, p) \in \Lambda(t)\}$, the size of this set is called the *(outgoing) degree* of the node and denoted by $\deg(p, t) = |\mathrm{nb}(p, t)|$. Note that for a system where every link is bidirectional $D(p, q, t) = D(q, p, t)$ and $q \in \mathrm{nb}(p, t) \Rightarrow p \in \mathrm{nb}(q, t)$ holds. The maximum resp. minimum degree of the whole graph is denoted as $\Delta(t) = \max_{p \in \Pi} \deg(p, t)$ resp. $\delta(t) = \min_{p \in \Pi} \deg(p, t)$.

2.3 Failure Model

Process crashes need not be clean, i.e., they may occur during a step, and thus it is possible that some but not all messages of this step are put onto the links. After a process has crashed it does not take any steps, i.e., messages sent to such a process are lost. We do not consider other types of process failures than crashes.

We define $C(t)$ to be the set of processes that have not crashed until t, and $\mathcal{C} = \bigcap_{t \in \mathcal{T}} C(t)$ to be the set of *correct* processes, i.e., processes that never crash. Conversely, let $F(t) = \Pi - C(t)$ to be the set of processes that have crashed by time t. $F(t)$ is also called the *failure pattern*. Finally, $\mathcal{F} = \bigcup_{t \in \mathcal{T}} F(t) = \Pi - \mathcal{C}$ is the set of *faulty* processes, i.e., processes that eventually crash.

Due to changes in the communication graph, the network may dynamically partition into components. In a system with only bidirectional links we make the following definitions: For a process p, the component $C(p, t)$ is defined as the subgraph of $G(t)$ that is induced by all nodes connected to p. If p has crashed by time t, we define $C(p, t) = \{p\}$. The final component of p, i.e., the component of p after the last crash in the system is denoted $C(p, \infty)$. Further, we group all processes in p's component by their distance

from p: $P_k(p,t) = \{q \in C(p,t) \mid D(p,q,t) = k\}$, and write $n_k(p,t) = |P_k(p,t)|$. Note that $P_1(p,t) = \mathrm{nb}(p,t)$.

By definition, a link only connects two non-crashed processes; thus we have $\forall t \in \mathcal{T}$: $\Lambda(t) \subseteq C(t) \times C(t)$. Further, if we assume only bidirectional links, the crash of a link from p to q induces obviously the crash of the link from q to p. A link can either be

- *lossy*: A message that is sent over this link may be lost
- *fair lossy*: Messages are classified into types. For every type, if infinitely many messages are sent over a fair lossy link, infinitely many of them are received.
- *reliable*: All messages that are sent are eventually received.
- *timely*: All messages that are sent are received according to some timing condition (see Section 2.5).

We have the following relations:

$$lossy \supseteq fair\ lossy \supseteq reliable \supseteq timely,$$

where $\supseteq$ denotes inclusion in the sense that a timely link fulfills the properties of a reliable link, a reliable link fulfills also the properties of a fair lossy link, etc.

A link could also be Byzantine faulty: There is no restriction on the behavior of such a link. A message sent over such a link may get correctly transmitted, lost, duplicated, and altered; the link may produce sporadic messages and does not fulfill any timing condition. Byzantine links are not considered in this thesis.

2.4 Execution Model

For $p \in \Pi$ denote by S_p the set of states of p. Let $m = |\Lambda|$ be the number of (unidirectional) links in the system. A *configuration* of the system is a vector of states of all processes together with m sets—one set for every link—of messages in transit on that link. A configuration is denoted by $C = (s_1, s_2, \cdots, s_n, L_{\lambda_1}, L_{\lambda_2}, \cdots, L_{\lambda_m})$ where L_{λ_j} is the set of messages on λ_j and $s_i \in S_i$.

Processes and the network operate by performing *steps*. A step can be one of the following:

multi message reception step (*mmr*): A multi message reception step includes reception of a non-empty set of messages, the computational step of the state machine and (optional) sending of messages. Formally, a multi message reception step is defined by a tuple $a = (p, s_p, R, S, s'_p)$, where $R = \{(msg_1^r, \lambda_1^r), \ldots, (msg_k^r, \lambda_k^r)\}$ and $S = \{(msg_1^s, \lambda_1^s), \ldots, (msg_\ell^s, \lambda_\ell^s)\}$, meaning p is in state s_p, p receives messages msg_i^r from links λ_i^r and sends messages msg_j^s over links λ_j^s, and s'_p is the state of p after

execution of this atomic step. We assume that processes are able to receive several messages from incoming links concurrently, thus the λ_i^r are not necessarily disjoint. The λ_j^s are defined to be disjoint, however. S may be empty.

A multi message reception step a is applicable to a configuration C, if p is in state s_p, and for all (msg_i^r, λ_i^r), $msg_i^r \in L_{\lambda_i^r}$. The configuration C' after a multi message reception step is the same as C, except that p is now in s'_p, for $1 \leq i \leq k$, msg_i^r is no more in $L_{\lambda_i^r}$, and every $L_{\lambda_j^s}$ also contains a message msg_j^s, for $1 \leq j \leq \ell$. This means, $C' = (s_1, \cdots, s'_p, \cdots, s_n, L'_{\lambda_1}, \cdots, L'_{\lambda_m})$ with $L'_\lambda = L_\lambda \cup \bigcup_{\lambda_i^s = \lambda} msg_i^s - \bigcup_{\lambda_i^r = \lambda} msg_i^r$.

message reception step (mr) : A message reception step includes reception of a single message, the computational step of the state machine and (optional) sending of messages. Formally, a message reception step is defined by a tuple $a = (p, s_p, (msg^r, \lambda^r), S, s'_p)$, with $S = \{(msg_1^s, \lambda_1^s), \ldots, (msg_\ell^s, \lambda_\ell^s)\}$, meaning p is in state s_p, p receives message msg^r from link λ^r and sends messages msg_j^s over links λ_j^s, and s'_p is the state of p after execution of this atomic step. The λ_j^s must be disjoint. S may be empty. A message reception step a is *applicable* to a configuration C, if p is in state s_p, and $msg^r \in L_{\lambda^r}$. The configuration C' after a message reception step is the same as C, except that p is now in s'_p, msg^r is no more in L_{λ^r}, and every $L_{\lambda_i^s}$ also contains a message msg_i^s. This means, $C' = (s_1, \cdots, s'_p, \cdots, s_n, L'_{\lambda_1}, \cdots, L'_{\lambda_m})$ with with $L'_\lambda = L_\lambda \cup \bigcup_{\lambda_i^s = \lambda} msg_i^s - \bigcup_{\lambda^r = \lambda} msg^r$. Note that this step is a special case of the multi message reception step with $|R| = 1$.

message loss step (ml): A message loss step models the situation when a lossy or fair lossy link drops a message sent over this link. Formally, a message loss step is a tuple $a = (\lambda, msg)$ where $\lambda \in \Lambda$ is a link and $msg \in L$ is a message. A message loss step $a = (msg, \lambda)$ is applicable to a configuration C, if $msg \in L_\lambda$ and λ is a lossy link. The configuration C' after a message loss step is the same as C, except that L_λ does no more contain msg. That is, $C' = (s_1, \cdots, s_n, L_{\lambda_1}, \cdots, L_\lambda - \{msg\}, \cdots, L_{\lambda_m})$.

local clock step (lc): A local clock step occurs if the local clock at a process takes a step and communicates this to the process. Formally, a local clock step is a tuple $a = (p, s_p, S, s'_p)$, where $S = set(msg_1^s, \lambda_1^s), \ldots, (msg_\ell^s, \lambda_\ell^s)$, meaning p is in state s_p, sends messages msg_j^s over links λ_j^s and is in state s'_p after the clock event. S may be empty. A local clock step a is applicable to a configuration C, if p is in s_p. For every state s_p there exists a local clock step that is applicable. The configuration C' after a local clock step is the same as C, except that p is now in s'_p and every $L_{\lambda_i^s}$ also contains a message msg_i^s. That is, $C' = (s_1, \cdots, s'_p, \cdots, s_n, L'_{\lambda_1}, \cdots, L'_{\lambda_m})$ with $L'_\lambda = L_\lambda \cup \bigcup_{\lambda_i^s = \lambda} msg_s^i$.

deadlock prevention step (dp): A deadlock prevention step at a process p is defined as a tuple $a = (p, s_p, S, s'_p)$, where $S = set(msg_1^s, \lambda_1^s), \ldots, (msg_\ell^s, \lambda_\ell^s)$, meaning p is in

state s_p, sends messages msg_j^s over links λ_j^s and is in state s'_p after the spontaneous deadlock prevention event. S may be empty. A deadlock prevention step a is applicable to a configuration C, if p is in s_p. For every state s_p there exists a deadlock prevention step that is applicable. The configuration C' after a message reception step is the same as C, except that p is now in s'_p and every $L_{\lambda_i^s}$ also contains a message msg_i^s. That is, $C' = (s_1, \cdots, s'_p, \cdots, s_n, L'_{\lambda_1}, \cdots, L'_{\lambda_m})$ with $L'_\lambda = L_\lambda \cup \bigcup_{\lambda_i^s = \lambda} msg_i^s$.

We denote the fact that a configuration C' results by applying a step a to a configuration C by $C \xrightarrow{a} C'$.

An *execution* $\sigma = (C_0, a_1, C_1, a_2, \ldots)$ is a (finite or infinite) sequence, which starts with some configuration C_0 and where, for every $i > 0$, a_i is applicable to C_{i-1} and results in C_i. An execution σ_1 is applicable to a finite execution σ_0, if the last configuration of σ_0 is equal to the first of σ_1. A *timed execution* is a sequence $\sigma = (C_0, a_1, t_1, C_1, a_2, t_2, \ldots)$, where $(C_0, a_1, C_1, a_2, \ldots)$ is an execution and $t_i \in \mathbb{R}$ is the real-time the step a_i occurs, with $t_i \leq t_{i+1}$ holds for all i. Let t_{GST} denote the time where our timing becomes correct in the following way: A timed execution is *timely*, if there is a *global stabilization time* t_{GST}, so that for every message msg that is sent by a step a_i there is some step a_j where msg is received, and $\tau^- \leq t_j - t_i \leq \tau^+$ if $t_i \geq t_{GST}$ and $t_j < t_{GST} + \tau^+$ if $t_i < t_{GST}$. That means, every message that is in transit at time t_{GST} is received before $t_{GST} + \tau^+$. How the bounds τ^+ and τ^- are fixed is described in Section 2.5.

Additionally, for the analysis of our algorithms, we define $v_p(t)$ to be the value of variable v at process p at time t before the step at time t, and if there is a step at t, we define $v'_p(t)$ to be the value of variable v at process p at time t after the step at time t.

We say a message msg is *in transit* from p to q at time t, if msg is in $L_{(p,q)}$ in the last configuration before t (note that this does not include messages sent at t). In other words, msg is in transit in the interval $(t_s, t_r]$, if it is sent in a step at t_s and received in a step at t_r. Further, we denote with $Q(p, q, t)$ the set of messages which are in transit from p to q or vice versa at time t, and with $Q(p, t) = \bigcup_{q \in \Pi} Q(p, q, t)$ all messages in transit from or to p. Consequently, $Q'(p, q, t)$ denotes the set of all messages from p to q or vice versa after a step at time t.

2.5 Timing Models

The definition of a timely execution requires only that every message that is sent at time t is received in the interval $[t + \tau^-, t + \tau^+]$. If there is synchrony in the system, which is the subject of this section, these parameters have to fulfill some condition. We are aware that there are some forms of synchrony that cannot be modeled by conditions on τ^+ and τ^- (e.g. ordering properties [MMR03]). However, for this work our approach is sufficient. Note that the overview of the timing models in this section is not a comprehensive one.

Some papers in literature [DLS88] distinguish between the time bounds of the network and the time bounds on the processing speeds of processes. In this work, we will consider the duration of the communication + transmission end-to-end delays which incorporates both, computing step times and transmission delays. Doing so not only simplifies analysis, but has also a higher *assumption coverage*[1].

To see this, assume $\mathbf{T}_c$ is the stochastic variable that describes the duration of a computational step (including message preparation and reception) and $\mathbf{T}_m$ is the stochastic variable that describes the duration of the transmission of a certain message. Let $p(t_c, t_m)$ be their joint probability density function and T_m resp. T_c the assumed bounds on $\mathbf{T}_m$ resp. $\mathbf{T}_c$, $\mathbf{T} = \mathbf{T}_c + \mathbf{T}_m$ be the stochastic variable that describes the total delay, and $T = T_m + T_c$ be the bound on $\mathbf{T}$. Then,

$$
\begin{aligned}
P[\mathbf{T} \leq T] = P[\mathbf{T}_m + \mathbf{T}_c \leq T] &= \iint\limits_{t_c + t_m < T} p(t_m, t_c)\ dt_c\, dt_m = \\
&= \int_0^{T_m}\int_0^{T_c} p(t_m, t_c)\ dt_c\, dt_m + \underbrace{\int_{T_m}^{T}\int_0^{T-t_m} p(t_m, t_c)\ dt_c\, dt_m}_{\geq 0} + \underbrace{\int_{T_c}^{T}\int_0^{T-t_c} p(t_m, t_c)\ dt_m\, dt_c}_{\geq 0} \geq \\
&\geq \int_0^{T_m}\int_0^{T_c} p(t_m, t_c)\ dt_c\, dt_m = P[\mathbf{T}_m \leq T_m \wedge \mathbf{T}_c \leq T_c],
\end{aligned}
$$

which clearly shows that the assumption coverage is higher with our approach.

2.5.1 Pure Asynchronous Model

This model makes no assumption on the transmission delay of a message and on the relative speeds of processes. Thus, $\tau^- = 0$ and $\tau^+ = \infty$[2]. Although this model has an assumption coverage of 1 regarding timing, it has the major drawback that many important agreement problems in distributed computing are not solvable in this model. For consensus in the presence of crashes this was shown in the seminal paper of Fischer, Lynch and Paterson in 1985 [FLP85], later for group membership [CHTCB96] and atomic commitment. However, if an asynchronous network is equipped with an appropriate

[1] All system models suffer from the fact, that their assumptions e.g. on timing or failure modes hold only with a certain probability in practice. The assumption coverage is the probability, that the assumptions hold also in a real system. Obviously, the assumption coverage depends on the model and the real implementation.

[2] Note that this does not imply that a message transmission can take infinitely long: $\tau^+ = \infty$ just means that there is no bound on the transmission delay. If the link is reliable, every message is received after finite time.

failure detector, these problems become solvable. From this it follows that these failure detectors are not implementable in the pure asynchronous model.

2.5.2 Synchronous Model

In the synchronous model $\tau^+ < \infty$ is known a priori by the algorithm. Further, $t_{GST} = 0$. This requires synchronous links and bounded relative process speeds as well. On the other hand, many problems are solvable in a simple and efficient way in this model. τ^- is usually assumed to be 0.

Another form of the synchronous model is that there is an a priori known bound on the jitter $\varepsilon = \tau^+ - \tau^-$. This variant is used in Chapter 5. Note that the measurement of absolute time values requires bounded drift clocks.

2.5.3 Partially Synchronous Models

Partial synchrony lies between pure asynchrony and pure synchrony. The partial synchronous model assumes $\tau^+ < \infty$, but this value is not known to the processes. Further t_{GST} may differ from 0 and is also unknown.

The original work [DLS88] distinguishes between an absolute upper bound Δ on message transmission delays (not end-to-end) and an upper bound Φ on the relative computational speeds of any two processes (which would not fit in our system model), and assumes either a known Δ *or* $t_{GST} > 0$. In later work [CT96] this is dropped.

An even weaker partial synchronous assumption is the existence of a finite average on the communication delay [FS04a, FS04b] (which also does not fit in our system model).

In [ADGFT03a] a model where only some links are eventually timely ($\diamond$-timely) is presented. A link is $\diamond$-timely if there is an unknown bound on the transmission delay that holds eventually, i.e., $t_{GST} > 0$ and $\tau^+ < \infty$ but unknown. All classical partial synchronous models require some sort of bounded drift clock.

2.5.4 The Θ-Model

The Θ-Model [LLS03] allows to define synchrony using time free semantics. No local clocks are needed in this model. It assumes that there is some bound $\Theta = \tau^+/\tau^- < \infty$, whereas τ^+ and τ^- need not to be known. Θ can either be known or unknown to the algorithms, and also $t_{GST} = 0$ and $t_{GST} > 0$ are variants of this model. The algorithms in Chapter 7 use the Θ-Model with unknown Θ.

2.6 Models used in this Thesis

Without going in detail, the models in the following chapters are classified using the terms defined in this Chapter.

	Chapter 4	**Chapter 5**	**Chapter 6**	**Chapter 7**
message-driven	no	no	no	yes
process crashes	yes	yes	yes	yes
link crashes	no	yes	yes	no
self-stabilizing	no	no	yes	yes
partitionable	no	yes	yes	yes
timing	part. synchronous	synchronous	part. synchronous	Θ
links	fair lossy unidirectional	reliable bidirectional	reliable bidirectional	reliable bidirectional
events	mr, lc, ml	mr,lc	mmr, lc	mmr,dp
t_{GST}	unknown	0	unknown	unknown

All models are assuming a sparsely connected network.

Chapter 3

Failure Detectors

A failure detector is a module located at each process in the distributed system that provides information about other processes. Although the original definition of Chandra and Toueg [CT96] does not restrict the output of a failure detector, we assume a failure detector outputs a list of processes. Despite their name, failure detectors in fact somehow encapsulate the synchrony of a network. Thus in an asynchronous system enhanced with a failure detector it is possible to solve problems that are otherwise unsolvable in a purely asynchronous system without a failure detector. However, there is still a gap between the synchrony of the system and the failure detector information: In [CBGS00] it is shown that an synchronous system is still stronger than an asynchronous system with a perpetual perfect failure detector $\mathcal{P}$.

The computational power of a failure detector depends also on the system it is used in: Failure detectors that may be reducible to each other in one system—like $\diamond\mathcal{P}$ and $\diamond\mathcal{Q}$ in a fully connected asynchronous system with reliable links—may exhibit a different relationship in another system—like a partitionable network. In the literature, most considerations about the computational power of a failure detector are made in the context of the reducibility by an asynchronous algorithm in the sense of FLP in a fully connected system with reliable links and crashing processes. We give a formal definition in Section 3.2.

This chapter gives an overview of failure detector definitions in literature and shows some of the relationships between them. It also introduces *local failure detectors*, which are a new class of failure detectors that can be employed in sparse networks. The chapter concludes with an outline of the applications of failure detectors, some considerations about the quality of service of a failure detector module and finally sketches some implementation principles. Applications of failure detectors are presented in Section 3.8.

3.1 Failure Detector Histories

In order to generalize the common failure detector definitions to partitionable networks of unknown size, we distinguish two types of failure detectors: If the output of the failure detector contains processes that are suspected to have crashed, we call the failure detector *suspicion based*. On the other hand, if it contains processes the failure detector does *not* suspect, we call it *trust based*.

As we just consider failure detectors that output lists of processes, we formally define a *failure detector history* to be a function $H(p,t) : \Pi \times \mathcal{T} \rightarrow 2^{\Pi}$. Additionally we define the *inverse failure detector history* $\tilde{H}(p,t) \triangleq \Pi - H(p,t)$. If a process q is in $H(p,t)$ (and thus not in $\tilde{H}(p,t)$) at some time t, we say p *suspects* q, else p *trusts* q.

A suspicion based failure detector outputs $H(p,t)$ at time t and process p, whereas a trust based failure detector outputs $\tilde{H}(p,t)$. The latter allows the failure detector to run in systems where it does not know all other processes in the system or even n a priori. For an application that communicates with a known subset of Π this information is equivalent to the one of a suspicion based failure detector.

3.2 Classes and Reducibility

To compare the distinct classes of failure detectors, in this section we define some relations. A lot of work in literature uses very sloppy terminology when comparing failure detectors. In most cases, with saying *failure detector $\mathcal{D}$ is weaker than $\mathcal{D}'$* they mean reducibility in a fully connected network with asynchronous reliable links and crashing processes only. But they do not say this.

Let $\mathcal{E}$ be an environment[1] that is characterized by a certain model from Chapter 2. An environment comprises:

- a topology model (e.g. fully connected, or sparse with given maximum degree Δ)
- an execution model, including the set of communication primitives and their semantics
- the source of event generation (time-driven or message-driven)
- a timing model (e.g. synchronous, partial synchronous, asynchronous)
- a failure model (e.g. f process crashes only, lossy links, allowed failure patterns, period of total state corruption)

[1] In [CHT96], an environment is defined as the set of all possible failure patterns. Note that we use a more general definition here.

Note that these items are not necessarily independent of each other. Consider further two failure detectors $\mathcal{D}$ and $\mathcal{D}'$. We say $\mathcal{D}'$ is *reducible in environment* $\mathcal{E}$ to $\mathcal{D}$ if there is a distributed transformation algorithm $T_{\mathcal{D}\to\mathcal{D}'}$ in $\mathcal{E}$, such that $T_{\mathcal{D}\to\mathcal{D}'}$ uses $\mathcal{D}$ and simulates $\mathcal{D}'$. We denote this fact by $\mathcal{D}' \preceq_{\mathcal{E}} \mathcal{D}$.

If $\mathcal{D}' \preceq_{\mathcal{E}} \mathcal{D}$ but not the inverse holds we write $\mathcal{D}' \prec_{\mathcal{E}} \mathcal{D}$. If $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$ and $\mathcal{D}' \preceq_{\mathcal{E}} \mathcal{D}$ holds we say $\mathcal{D}$ and $\mathcal{D}'$ are *equivalent in* $\mathcal{E}$ and denote this by $\mathcal{D} \cong_{\mathcal{E}} \mathcal{D}'$. If neither $\mathcal{D} \preceq_{\mathcal{E}} \mathcal{D}'$ nor $\mathcal{D}' \preceq_{\mathcal{E}} \mathcal{D}$ holds, they are *incomparable*, denoted $\mathcal{D}' \prec\succ_{\mathcal{E}} \mathcal{D}$

A stronger relationship is proper inclusion. We say $\mathcal{D}$ *includes* $\mathcal{D}'$ if the properties of $\mathcal{D}$ are fulfilled by the properties of $\mathcal{D}'$, or can be obtained by a local function[2]. We denote this by $\mathcal{D}' \subseteq \mathcal{D}$, resp. $\mathcal{D} \supseteq \mathcal{D}'$. If $\mathcal{D}' \subseteq \mathcal{D}$ but $\mathcal{D}' \not\subseteq \mathcal{D}$ we write $\mathcal{D}' \subset \mathcal{D}$; if $\mathcal{D}' \subseteq \mathcal{D}$ and $\mathcal{D}' \subseteq \mathcal{D}$ we write $\mathcal{D}' = \mathcal{D}$. Note that inclusion implies reducibility in any environment, but in general the inverse does not hold:

$$\mathcal{D} \supseteq \mathcal{D}' \Rightarrow \mathcal{D} \preceq \mathcal{D}' \tag{3.1}$$

In particular, we consider the environment $\mathcal{E}_A$, which is the time-driven system with asynchronous, reliable links in a fully connected network as described in [FLP85]. Most work in the literature uses this environment for reduction, thus we omit the index in $\mathcal{E}_A$ and mean this environment, if not mentioned otherwise. Some other environments considered in this thesis are the partitionable asynchronous model $\mathcal{E}_P$ and the asynchronous message-driven self-stabilizing model $\mathcal{E}_S$.

3.3 Classical Failure Detectors

Chandra and Toueg have defined some classes of failure detectors that have been shown to be very useful for solving several fundamental problems in fault-tolerant distributed computing. All of their classes of failure detectors are defined via a completeness property and an accuracy property. Intuitively, the completeness property requires the failure detector to suspect faulty processes whereas the accuracy property prevents it from the trivial solution of suspecting all (correct) processes by requiring that some processes are not suspected. In the following we review the failure detector definitions by Chandra and Toueg [CT96]:

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

$$\exists t_0, \forall p \in \mathcal{F}, \forall q \in \mathcal{C}, \forall t \geq t_0 : \ p \in H(q,t)$$

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

$$\exists t_0, \forall p \in \mathcal{F}, \exists q \in \mathcal{C}, \forall t \geq t_0 : \ p \in H(q,t)$$

[2]Note that by such a function no messages are exchanged.

The completeness properties both require things to be *eventually* done. A failure detector that would know all faulty processes from the beginning would be able to predict the future and is thus not implementable in real systems. However, for theoretical analysis such a failure detector called Marabout ($\mathcal{M}$) has indeed been defined (see Section 3.7.1 for details). Accuracy properties, however, may hold either right from the beginning (perpetually) or eventually:

Strong Accuracy. No process is suspected before it crashes. Formally,

$$\forall t, \forall p, q \in C(t) : \ p \notin H(q, t)$$

Weak Accuracy. Some correct process is never suspected. Formally,

$$\exists p \in \mathcal{C}, \forall t, \forall q \in C(t) : \ p \notin H(q, t)$$

Eventually Strong Accuracy. There is a time after which correct processes are not suspected by any correct process. Formally,

$$\exists t_0, \forall t \geq t_0, \forall p, q \in C(t) : \ p \notin H(q, t)$$

Eventually Weak Accuracy. There is a time after which some correct process is never suspected by any correct process. Formally,

$$\exists t_0, \exists p \in \mathcal{C}, \forall t \geq t_0, \forall q \in \mathcal{C} : \ p \notin H(q, t)$$

From these properties, Chandra and Toueg derived the following classes of failure detectors:

	Strong Completeness	Weak Completeness
Strong Accuracy	$\mathcal{P}$ (*perfect*)	$\mathcal{Q}$
Weak Accuracy	$\mathcal{S}$ (*strong*)	$\mathcal{W}$ (*weak*)
Eventual Strong Accuracy	$\diamond\mathcal{P}$ (*eventually perfect*)	$\diamond\mathcal{Q}$
Eventual Weak Accuracy	$\diamond\mathcal{S}$ (*eventually strong*)	$\diamond\mathcal{W}$ (*eventually weak*)

By definition, we have the following hierarchy of relations:

$$\mathcal{W} \supset \mathcal{Q} \qquad \mathcal{W} \supset \mathcal{S} \qquad \mathcal{Q} \supset \mathcal{P} \qquad \mathcal{S} \supset \mathcal{P} \tag{3.2}$$

$$\diamond\mathcal{W} \supset \diamond\mathcal{Q} \qquad \diamond\mathcal{W} \supset \diamond\mathcal{S} \qquad \diamond\mathcal{Q} \supset \diamond\mathcal{P} \qquad \diamond\mathcal{S} \supset \diamond\mathcal{P} \tag{3.3}$$

$$\diamond\mathcal{W} \supset \mathcal{W} \qquad \diamond\mathcal{Q} \supset \mathcal{Q} \qquad \diamond\mathcal{S} \supset \mathcal{S} \qquad \diamond\mathcal{P} \supset \mathcal{P} \tag{3.4}$$

In [CT96] it has been shown that weak completeness can be reduced to strong completeness by an asynchronous algorithm in a fully connected network with reliable links. Thus

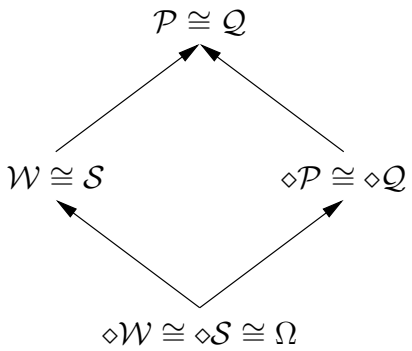

Figure 3.1: The hierarchy of classical failure detectors including Ω. An arrow from $\mathcal{A}$ to $\mathcal{B}$ means $\mathcal{A}$ is purely weaker than $\mathcal{B}$.

in a non-partitionable network and if no self-stabilization is required, the classes $\mathcal{P}$ and $\mathcal{Q}$ (resp. $\mathcal{S}$ and $\mathcal{W}$ and their eventual variants) are equivalent:

$$\mathcal{P} \cong \mathcal{Q} \qquad \mathcal{S} \cong \mathcal{W} \qquad \diamond\mathcal{P} \cong \diamond\mathcal{Q} \qquad \diamond\mathcal{S} \cong \diamond\,\mathcal{W} \tag{3.5}$$

Among these, all eventual failure detector classes ($\diamond\mathcal{W}$, $\diamond\mathcal{S}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{P}$) are implementable in a classical partial synchronous system [DLS88, CT96], whereas this is not possible for perpetual failure detectors ($\mathcal{W}$, $\mathcal{S}$, $\mathcal{Q}$, $\mathcal{P}$) [LFA02]. They require a system that allows reliable detections, i.e., timing assumptions that hold from the beginning and are known to the processes. Most implementation of these classes thus require a synchronous system.

3.4 The Weakest Failure Detector for Consensus

In [CHT96] a failure detector called Ω is presented. It is a trust based failure detector that outputs exactly one process at every point in time. An Ω failure detector fulfills the following property:

Eventual Leadership There is a time after which all the correct processes always trust the same correct process. Formally,

$$\exists t_0, \exists p \in \mathcal{C}, \forall q \in \mathcal{C}, \forall t \geq t_0 : \ \tilde{H}(q,t) = \{p\}$$

Obviously every Ω failure detector is a special case of an eventually weak failure detector. Moreover, $\diamond\mathcal{W}$ is reducible to Ω in $\mathcal{E}_A$, which has also been shown to be the weakest failure detector that solves consensus [CHT96]:

$$\Omega \cong \diamond\mathcal{W} \cong \diamond\mathcal{S} \tag{3.6}$$

More recently [ADGFT03a, ADGFT03b], a system has been identified that allows the implementation of Ω but not $\diamond\mathcal{S}$: It comprises fair lossy links and only some links of the system have to be eventually timely. We use this model also for our algorithms in Chapter 4. This shows that although Ω is strong enough to solve consensus, it can be implemented with very weak synchrony and reliability assumptions of the network.

Another issue raised in [ADGFT01] is *stability*. The eventual leadership property of Ω requires only eventual stabilization. This still allows an implementation to change the leader even if the current leader is correct and has been selected by all processes for a long time. In practice, for a badly designed failure detector algorithm this could be the consequence of a change of the timing of the network. Obviously such a behavior is not desirable, since it often causes overhead at application processes that have to react to such changes. Therefore, a previously agreed leader should only be replaced if it crashes. A failure detector with such a property is called *stable*. Formally, a *k-stable* algorithm guarantees that in every run, if p is leader at time t and p does not crash during $[t-k\tau^+, t]$ then p is leader at least until it crashes.[3]

3.5 Failure Detectors for Partitionable Networks

We can extend the strong properties of classical failure detectors to partitionable networks [DFKM97, ACT99, FKM+95] in a natural way:

Strong Completeness. For any two processes that become disconnected (this includes the case that one of them crashed), there is a time after which they permanently suspect each other. Formally,

$$\exists t_0 \forall t \geq t_0 : \ q \notin C(p,t) \Rightarrow \exists t_1 \forall t \geq t_1 : \ q \notin \tilde{H}(p,t)$$

Strong Accuracy. No two processes that are permanently connected suspect each other. Formally,

$$\forall t : \ q \in C(p,t) \Rightarrow \forall t' : \ q \in \tilde{H}(p,t')$$

Eventual Strong Accuracy. For any two processes that are permanently connected, there is a time after which they permanently do not suspect each other. Formally,

$$\exists t_0 \forall t \geq t_0 : \ q \in C(p,t) \Rightarrow \exists t_1 \forall t \geq t_1 : \ q \in \tilde{H}(p,t)$$

Note that such a natural generalization is not possible for weak properties, since the connectedness condition is defined pairwise between processes. It is possible, however, to

[3] The original definition "a *k-stable* algorithm guarantees that in every run, if p is leader at time t and p does not crash during $[t-k\tau^+, t+1]$ then p is leader at time $t+1$" has been adapted to our continuous time model

use a phrase like "in every component there is a process that is not suspected by all other processes in the component". We abstain from a formal definition for such properties.

We call a failure detector for a partitionable system *eventually perfect* if it fulfills strong completeness and eventual strong accuracy. The class of eventually perfect failure detectors for partitionable networks is denoted by $\diamond\mathcal{P}_p$. Note that this definition is a true generalization of failure detectors for non-partitionable systems: If the network does not partition, then the definitions are equivalent. With such a failure detector, some a priori known subset of processes can solve consensus if a majority of them remains connected and there is a communication module that implements reliable point to point connections between them [ACT99]. It can also be used to solve partitionable group membership [BDM01].

A failure detector for a partitionable system is called *perfect*, if it fulfills strong completeness and strong accuracy. The class of perfect failure detectors for partitionable networks is denoted by $\mathcal{P}_p$.

	Strong Completeness
Strong Accuracy	$\mathcal{P}_p$ (*partitional perfect*)
Eventual Strong Accuracy	$\diamond\mathcal{P}_p$ (*partitional eventually perfect*)

3.6 Local Failure Detectors

For sparse networks, we now define properties that lie between the strong and weak properties of classical failure detectors:

Local Completeness. Eventually every process that crashes is permanently suspected by every correct neighbor. Formally,

$$\forall p \in \mathcal{F}\ \forall q \in \mathrm{nb}(p) \cap \mathcal{C} :\ \exists t_0 \forall t \geq t_0\ p \in H(q,t)$$

Local Accuracy. No correct processes is ever suspected by any neighbor. Formally,

$$\forall p \in \mathcal{C}\ \forall q \in \mathrm{nb}(p) \cap \mathcal{C} :\ \exists t_0 \forall t \geq t_0\ p \notin H(q,t)$$

Eventually Local Accuracy. There is a time, after which correct processes are not suspected by any correct neighbor. Formally,

$$\forall p \in \mathcal{C}\ \forall q \in \mathrm{nb}(p) \cap \mathcal{C} :\ \exists t_0 \forall t \geq t_0\ p \notin H(q,t)$$

From these properties the following classes of failure detectors result:

	Local Completeness
Local Accuracy	$\mathcal{P}_\ell$ (*local perfect*)
Eventual Local Accuracy	$\diamond\mathcal{P}_\ell$ (*eventually local perfect*)

We give self-stabilizing implementations for $\diamond\mathcal{P}_\ell$ in Section 7.4 and Section 7.5. The following relations follow directly from the definitions:

$$\mathcal{P}_\ell \supseteq \mathcal{P} \qquad\qquad \diamond\mathcal{P}_\ell \supseteq \diamond\mathcal{P} \tag{3.7}$$

Remarks:

- The definition of local failure detectors for partitionable and non-partitionable networks are equivalent, since if two processes get partitioned from another, they also loose their neighborhood relation.
- For non-partitionable networks, a failure detector from $\diamond\mathcal{P}_\ell$ can easily be transformed to a global eventually perfect failure detector $\diamond\mathcal{P}$ by an asynchronous algorithm. For a partitionable network $\diamond\mathcal{P}_p$ can be reduced to $\diamond\mathcal{P}_\ell$. Transformation from local to global failure detectors is addressed in Sections 6.5 and 7.8 .

3.7 Other Classes of Failure Detectors

3.7.1 The Future Predicting Marabout

A failure detector of only theoretical interest has been introduced by Guerraoui [Gue01]. The Marabout failure detector $\mathcal{M}$ fulfills the properties:

Perpetual Completeness. Every faulty process is permanently suspected by every correct process. Formally,

$$\forall p \in \mathcal{F}\ \forall q \in \mathcal{C} :\ \forall t\ p \in H(q,t)$$

Perpetual Accuracy.

$$\forall p, q \in \mathcal{C} :\ \forall t\ p \notin H(q,t)$$

In contrast to $\mathcal{P}$, which provides perfect failure detection, this failure detector provides perfect failure prediction. Since no real failure detector can predict the future, this failure detector is not implementable in realistic settings.

Guerraoui showed that $\mathcal{P}$ and $\mathcal{M}$ are incomparable, although both can be used to solve *terminating reliable broadcast*, *non-blocking atomic commitment* and *leader election*[4].

[4]This may not be confused with the eventual leader election, as provided by Ω. Leader election in this sense may not make any mistake [SM95]

3.7.2 Perfect Failure Detectors of Larrea

Larrea [Lar02] relates $\mathcal{M}$ and $\mathcal{P}$ by introducing a whole family of perfect failure detectors via four different accuracy properties.

Strong Accuracy 1. No process is suspected before it crashes. Formally,

$$\forall t, \forall p, q \in C(t) : \ p \notin H(q,t)$$

Strong Accuracy 2. No process is suspected by any correct process before it crashes. Formally,

$$\forall t, \forall p \in C(t), \forall q \in \mathcal{C} : \ p \notin H(q,t)$$

Strong Accuracy 3. No correct process is ever suspected. Formally,

$$\forall t, \forall p \in \mathcal{C} \ \forall q \in \Pi : \ p \notin H(q,t)$$

Strong Accuracy 4. No correct process is ever suspected by any correct process. Formally,

$$\forall t, \forall p, q \in \mathcal{C} : \ p \notin H(q,t)$$

Strong accuracy 1 equals the classical strong accuracy property of Section 3.3. By combining each of these accuracy properties with strong completeness, we obtain four classes $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, and $\mathcal{P}_4$. By definition $\mathcal{P}_1 = \mathcal{P}$. Larrea shows the following relationships:

$$\mathcal{P}_2 \prec \mathcal{P}_1 \qquad \mathcal{P}_3 \prec \mathcal{P}_1 \qquad \mathcal{P}_4 \prec \mathcal{P}_2 \qquad \mathcal{P}_4 \prec \mathcal{P}_3 \tag{3.8}$$

$$\mathcal{P}_3 \prec\!\succ \mathcal{P}_2 \qquad \mathcal{P}_3 \prec \mathcal{M} \qquad \diamond\mathcal{S} \prec \mathcal{P}_4 \tag{3.9}$$

All failure detectors $\mathcal{P}_1$, $\mathcal{P}_2$, $\mathcal{P}_3$, and $\mathcal{P}_4$ can be implemented in a synchronous system. Since $\diamond\mathcal{S} \prec \mathcal{P}_4$, we can use $\mathcal{P}_4$ to solve uniform consensus. Further, $\mathcal{P}_4$ suffices to solve non-blocking atomic commitment and $\mathcal{P}_3$ suffices to solve terminating reliable broadcast.

3.7.3 Heartbeat Failure Detector

The heartbeat failure detector [ACT99] differs from all other previously described failure detectors in that it outputs not a list of processes but a list of non-negative integers, one for each other process in the system. We denote with $H(p,t)[q]$ the integer for process q in the failure detector history of p. It is defined for a partitionable system with link crashes.

HB-Completeness. At each correct process p, the heartbeat sequence of every process not in the partition of p is bounded. Formally,

$$\forall p \in \mathcal{C}, \forall q \in C(p,\infty), \exists K \in \mathbb{N}, \forall t : \ H(p,t)[q] \leq K$$

HB-Accuracy. Comprises the two subproperties:

- At each process p, the heartbeat sequence of every process is nondecreasing. Formally,
$$\forall p, q \in \Pi, \forall t_1 \leq t_2 : \ H(p, t_1)[q] \leq H(p, t_2)[q]$$
- At each correct process p, the heartbeat sequence of every process in the partition of p is unbounded. Formally,
$$\forall p \in \mathcal{C}, \forall q \in C(p, \infty), \forall K \in \mathbb{N}, \exists t : \ H(p, t)[q] > K$$

The heartbeat failure detector can be implemented in a purely asynchronous system and is thus not sufficient to solve consensus or any stronger agreement problem. However, it can be used for quiescent[5] reliable communication.

It is important not to confuse this class of failure detectors with *heartbeat-style* failure detectors, i.e., failure detectors that use heartbeats as implementation principle (see Section 3.10).

3.7.4 The Θ Failure Detector

In [ATD99] a failure detector called Θ[6] has been formalized. It is shown to be the weakest failure detector for uniform reliable broadcast in a system with process crashes and fair lossy links.

Θ-Completeness. There is a time after which correct processes do not trust any process that crashes. Formally,
$$\exists t_0, \forall p \in \mathcal{C}, \forall q \in \mathcal{F}, \forall t \geq t_0 : \ q \notin \tilde{H}(p, t)$$

Θ-Accuracy. If there is a correct process then, at every time, every process trusts at least one correct process. Formally,
$$\mathcal{F} \neq \Pi \Rightarrow \forall t, \forall p \in \Pi, \exists q \in \mathcal{C} : \ q \in \tilde{H}(p, t)$$

Note that the correct process trusted by a process p may change infinitely often, and it is not necessarily the same as the correct process trusted by another process q.

[5] A reliable broadcast algorithm is *quiescent*, if it sends only a finite number of messages when broadcast is invoked a finite number of times [ACT99].

[6] Not to be confused with the homonymous timing model of [LLS03, SW05]

3.7.5 Failure Detectors for the Crash-Recovery Model

In [ACT00] and [HMR98] a system model that allows processes to recover after a crash is considered. Here we have to distinguish not only between correct and faulty processes but also between those that crash and recover an infinite number of times and others that do not. Thus, a process p can be classified as follows:

Always-up: Process p never crashes.

Eventually-up: Process p crashes at least once, but there is a time after which p is permanently up.

Eventually-down: There is a time after which process p is permanently down.

Unstable: Process p crashes and recovers infinitely many times.

A process is said to be *good* if it is either always-up or eventually-up, else it is *bad*.

The failure detector in [HMR98] outputs a list of processes and has the property:

Strong Completeness. Eventually every bad process is permanently suspected by all good processes.[7]

As the authors of [ACT00] show, such a completeness definition is not very useful, since every implementation inevitably has runs where all processes are good and nevertheless some processes are suspected forever, even in synchronous systems. As a solution they propose a new class of failure detectors that can informally been seen as a combination of $\diamond\mathcal{S}$ and the heartbeat failure detector. At each process p, the output of such a failure detector consists of a list of trusted processes and an epoch counter for each process in this list. Intuitively, a process q is in the list of trusted processes if p believes that q is currently up, and the epoch counter is p's estimate of how often q has crashed and recovered so far.

$\diamond\mathcal{S}_e$ is the class of failure detectors that satisfies the following properties:

Monotonicity. At every good process, eventually the epoch numbers are nondecreasing.

Completeness. For every bad process b and for every good process g, either eventually g permanently suspects b or b's epoch number at g is unbounded.

Accuracy. For some good process p and for every good process g, eventually g permanently trusts p and p's epoch number at g stops changing.

The authors also define a stronger accuracy property:

[7] We abstain from a formal definition here since the crash-recovery model is not covered by our prior formal considerations.

Strong Accuracy. For some good process p: (a) for every good process g, eventually g permanently trusts p and p's epoch number at g stops changing; and (b) for every unstable process u, eventually whenever u is up, u trusts p and p's epoch number at u stops changing.

Intuitively this property requires also the failure detector at unstable processes to behave correctly whenever they are up. The class of failure detectors that fulfill monotonicity, completeness and strong accuracy is denoted by $\diamond\mathcal{S}_u$. Obviously, $\diamond\mathcal{S}_u$ is stronger than $\diamond\mathcal{S}_e$. In systems where a majority of processes are good, $\diamond\mathcal{S}_e$ can be transformed into $\diamond\mathcal{S}_u$.

If the number of always-up processes is larger than the number of bad processes consensus can be solved using $\diamond\mathcal{S}_e$ even without stable storage. If a majority of processes is good, consensus can be solved with stable storage and $\diamond\mathcal{S}_u$.

3.7.6 Γ-Accurate failure detectors

The classical failure detectors of Chandra and Toueg monitor all processes in the system: The completeness properties require to eventually suspect every crashed process and the accuracy properties restrict the failures that can be made by all processes in the system.

A Γ-accurate failure detector [GS96] restricts the accuracy property to a subset Γ of processes. In more detail, for a given set $\Gamma \subseteq \Pi$ they guarantee:

Γ-Strong Accuracy. No process in Γ is suspected by any process in Γ before it crashes. Formally,

$$\forall t, \forall p, q \in C(t) \cap \Gamma : \ p \notin H(q, t)$$

Γ-Weak Accuracy. Some correct process (not necessarily in Γ) is never suspected by any process in Γ. Formally,

$$\exists p \in \mathcal{C}, \forall t, \forall q \in C(t) \cap \Gamma : \ p \notin H(q, t)$$

Together with similar extensions for the eventual accuracy properties, we obtain classes $\mathcal{P}(\Gamma)$, $\mathcal{S}(\Gamma)$, $\mathcal{W}(\Gamma)$, $\diamond\mathcal{P}(\Gamma)$, $\diamond\mathcal{S}(\Gamma)$, and $\diamond\mathcal{W}(\Gamma)$. They relate to the classical failure detector classes as follows:

- If $|\Gamma| > n/2$ and $f < n/2$, then $\diamond\mathcal{S}(\Gamma)$ can be transformed into $\diamond\mathcal{S}$ in a system with eventually reliable channels.
- If $\Gamma \subset \Pi$, even with reliable channels $\diamond\mathcal{W}(\Gamma)$ cannot be transformed into $\diamond\mathcal{W}$. Since $\diamond\mathcal{W}$ is the weakest failure detector for consensus, $\diamond\mathcal{W}(\Gamma)$ does not suffice to solve consensus.
- If $\Gamma \subset \Pi$, even with reliable channels $\mathcal{P}(\Gamma)$ cannot be transformed into $\mathcal{P}$.

Thus we have:

$$\diamond\mathcal{S} \cong_{\mathcal{E}_{Maj}} \diamond\mathcal{S}(\Gamma) \qquad \diamond\mathcal{W}(\Gamma) \prec \diamond\mathcal{W} \qquad \mathcal{P}(\Gamma) \prec \mathcal{P} \tag{3.10}$$

where $\mathcal{E}_{Maj}$ is the environment with a majority of correct processes and eventual reliable links as described above.

3.7.7 Limited Accuracy Failure Detectors

A slightly weaker accuracy property than the Γ-accuracy is given in [MR00, MR99b]: The set of processes for which the accuracy property needs to hold is not given a priori but only the size is known.

x-Accuracy. There is a set $Q \subseteq \Pi$ of size x, such that no process in Q is suspected by any process in Q before it crashes. Formally,

$$\exists Q \subseteq \Pi, \forall t, \forall p, q \in C(t) \cap Q : \ p \notin H(q,t) \wedge |Q| = x$$

Eventually x-Accuracy. There is a set $Q \subseteq \Pi$ of size x, such that there is a time, after which correct processes from Q are not suspected by any correct process in Q. Formally,

$$\exists Q \subseteq \Pi, \exists t_0, \forall t \geq t_0, \forall p, q \in C(t) \cap Q : \ p \notin H(q,t) \wedge |Q| = x$$

By combining these accuracy properties with the strong completeness of Chandra and Toueg, we obtain the classes $\mathcal{S}_x$ and $\diamond\mathcal{S}_x$. By definition, $\mathcal{S}_n$ is $\mathcal{S}$, whereas $\mathcal{S}_1$ does not provide any information about other processes and does not enhance an asynchronous system. A failure detector of class $\mathcal{S}_x$ (resp. $\diamond\mathcal{S}_x$) can be used to solve the k-set agreement problem [Cha90], if $f < k + x - 1$ resp. $f < \min(n - k\lfloor n/(k+1)\rfloor, k + x - 1)$.

3.7.8 Eventual Consistent Failure Detector

Larrea et al. [LFA01] introduce a failure detector called *eventually consistent failure detector* $\diamond\mathcal{C}$ which fulfills strong completeness and the following accuracy property:

Eventually Consistency Accuracy. There is a deterministic function $leader : \ 2^{\Pi} \rightarrow \Pi$, a time t and a correct process p such that for all times $t' > t$, for every correct process q, $p \notin H(q,t')$ and $leader(\Pi - H(q,t')) = p$.

The authors show that

$$\diamond\mathcal{C} \cong \Omega \tag{3.11}$$

whereas

$$\diamond\mathcal{S} \subseteq \diamond\mathcal{C} \subseteq \diamond\mathcal{P} \tag{3.12}$$

holds by definition. A failure detector from $\diamond\mathcal{C}$ can be implemented as efficient as a failure detector from $\diamond\mathcal{S}$, but allows a more efficient consensus algorithm that does not use the rotating coordinator paradigm and reaches consensus in one round after stabilization.

3.7.9 Failure Detectors for Byzantine Faults

A natural generalization of the concept of failure detectors to Byzantine faults is not easy if we want to keep such a definition implementable: First of all, there are Byzantine faults that are undetectable for a single process for various reasons. Secondly, such an implementation of a failure detector for Byzantine faults must always depend on the algorithm that uses the failure detector, since the message pattern and the expected content is obviously not the same for all algorithms.

Thus, all approaches regarding Byzantine failure detectors use the concept of monitoring the messages the application process sends and receives. The first approach, called *muteness failure detectors* [DS98, DGGS99, DGG02] assumes a round based message pattern and requires the application process to feed the failure detector with all information it needs. Further, this failure detector is not a distributed oracle but a local module that communicates only with the application process. The second approach, called *Byzantine fault detector* [KMMS03] is restricted to a certain consensus algorithm and is also only a local module. Both approaches use timeouts on the perceived message pattern to calculate a list of suspects.

Every implementation of a failure detector for Byzantine faults inherently depends on a certain application and thus cannot be independently used by more than one application process. Further, only detectable faults can be encapsulated by the failure detector, the application process still has to deal with undetectable value faults. For these reasons, using the concept of failure detectors for Byzantine faults seems not to be very auspicious.

3.8 Applications of Failure Detectors

Many problems that are unsolvable in a purely asynchronous system are solvable if the system is augmented with an appropriate failure detector. Without aiming for completeness, we show the most popular problems that can be solved in asynchronous systems with a failure detector.

Note that there are still problems that require a synchronous system, like *clock synchronization* [Wid03, Wid04]. Further, there are problem that require information that is not only depending on the failure pattern: The problem of *predicate detection* [GP01, GP02] requires additional information about the *state* of the channel, and is thus unsolvable by

any failure detector. A *failure detector sequencer* is a pure generalization of a failure detector and provides sufficient and necessary information to the processes to solve predicate detection [GP02].

Consensus. The consensus problem has already been introduced as a motivation for failure detectors in Section 1.1. The work of Chandra and Toueg [CT96] has spawned a considerable amount of research [AT03, vR03, MR99a, MR01] on how consensus can be implemented using failure detectors.

Consensus algorithms atop of the eventual strong failure detector $\diamond\mathcal{S}$ are often built using the *rotating coordinator paradigm* [CM84, DLS88]. Such an algorithm proceeds in asynchronous rounds. In every round r process $c = (r \mod n) + 1$ is the coordinator. All processes send their estimate to the current coordinator and wait either for the answer or that the failure detector suspects the coordinator. The coordinator waits for $(n+1)/2$ estimates, carefully selects one of them and broadcasts this value back to all processes. In a second acknowledgment phase the coordinator detects whether agreement has been achieved or not and in case, reliably broadcasts the decision value. Ω based consensus algorithms work similarly, here the coordinator does not rotate but is always the current leader. In both cases, eventually every process trusts a coordinator and thus agreement can be achieved. To solve consensus with an eventual failure detector ($\diamond\mathcal{P}$, $\diamond\mathcal{Q}$, $\diamond\mathcal{S}$, $\diamond\mathcal{W}$, Ω) a majority of correct processes is needed ($f < n/2$).

Consensus using a perpetual failure detector can be solved for $f < n$. The algorithm proposed by Chandra and Toueg proceeds in asynchronous rounds, where the failure detector is used to avoid that a process waits infinitely long for a faulty processes. Like for a synchronous algorithms [AW98], the algorithm exchanges a vector of estimates of the initial values of the processes for $f + 1$ rounds.

Group Membership. Group membership is the base for many *group communication systems* [CKV01]. A group communication system comprises services like *virtual synchrony*, *atomic broadcast*, *total order broadcast*, etc. The definitions for such services differ very much, and so do the definitions of group membership.

Informally, a group membership service provides the processes with an agreed set of correct processes. In some specifications, processes may voluntarily leave a group, and new processes may join a group. Group communication systems often consider partitionable systems, so there are definitions for partitionable group membership as well as for *primary component group membership*.

Failure detectors serve two purposes in group membership: First they provide information which processes are correct—here we need a perfect or eventually perfect failure detector—and are needed to solve the inherent agreement problem in group membership—here an Ω failure detector suffices.

Atomic Broadcast. Informally speaking, atomic broadcast requires all correct processes to deliver all messages in the same order. This requirement corresponds to agreement on the delivery order and thus —not surprisingly—consensus and atomic broadcast are reducible to each other in an asynchronous system [CT96].

More formally, atomic broadcast comprises two primitives, *broadcast* and *deliver*. The properties of atomic broadcast [HT93, CT96] are the ones of *reliable broadcast*[8]

(Validity.) If a correct process broadcasts a message m, then it eventually delivers m.

(Agreement.) If some correct process delivers a message m, then all correct processes eventually deliver m.

(Uniform Integrity.) For every message m, every process delivers m at most once, and only if m was previously broadcast by the sender of m.

together with the property

(Total Order.) If correct 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'.

Because of its relation to consensus, the weakest failure detector for atomic broadcast is Ω.

Uniform Reliable Broadcast. Uniform reliable broadcast [HT93, ATD99] is a communication primitive that requires that if a process delivers a message then all correct processes also deliver this message. In contrast to reliable broadcast, the agreement property also includes messages delivered at faulty processes. Thus for the two communication primitives *broadcast* and *deliver* we have the following properties:

(Validity.) If a correct process broadcasts a message m, then it eventually delivers m.

(Uniform Agreement.) If some process delivers a message m, then all correct processes eventually deliver m.

(Uniform Integrity.) For every message m, every process delivers m at most once, and only if m was previously broadcast by the sender of m.

The weakest failure detector for uniform reliable broadcast is the failure detector Θ, cf. Section 3.7.4, [ATD99].

[8]Note that reliable broadcast itself can be solved without any failure detector in an asynchronous system.

Terminating Reliable Broadcast. Terminating reliable broadcast is reliable broadcast with an additional property, which requires that correct processes always deliver a message. Also, the integrity property is slightly weakened, to allow processes to deliver SF, which is a special message that indicates that the sender was faulty. Thus we have the following properties for terminating reliable broadcast [HT93]:

(Termination.) Every correct process eventually delivers some message.

(Validity.) If the sender is correct and broadcasts a message m, then all correct processes eventually deliver m.

(Agreement.) If a correct process delivers a message m, then all correct processes eventually deliver m.

(Integrity.) Every correct process delivers at most one message, and if it delivers $m \neq \text{SF}$ then the sender must have broadcast m.

Terminating reliable broadcast can be solved in asynchronous systems using $\mathcal{P}$, but not with $\mathcal{S}$, $\diamond\mathcal{P}$ or $\diamond\mathcal{S}$ [CT96].

Non-blocking Atomic Commitment. The non-blocking atomic commitment problem [Gue02, Ske81] requires all processes to reach a common decision, *commit* or *abort*, according to some initial votes, *yes* or *no*, such that the following properties hold:

(Agreement.) No two processes decide differently.

(Termination.) Every correct process eventually decides.

(Abort-Validity.) Abort is the only possible decision if some process votes *no*.

(Commit-Validity.) Commit is the only possible decision if every process is correct and votes *yes*.

Non-blocking atomic commitment can be solved in asynchronous systems using $\mathcal{P}$, but not with $\mathcal{S}$, $\diamond\mathcal{P}$ or $\diamond\mathcal{S}$ [CT96]. However, this does not imply that non-blocking atomic commitment is harder to solve than consensus, since [Gue02] showed that Consensus and non-blocking atomic commitment are incomparable.

3.9 The Quality of Service of Failure Detection

In [CTA00] a comprehensive list of metrics is defined that characterize the quality of service of a failure detector. These metrics are defined for systems with probabilistic behavior and thus only partially apply to our system model. Their system model assumes

that messages get lost on a link with a given message loss probability and the message delay is given by a random variable $\mathbf{T}$ with finite expected value $E\{\mathbf{T}\}$ and variance $var\{\mathbf{T}\}$.

They define three *primary metrics* based on a two processes system comprising processes p and q:

Detection time $\mathbf{T}_D$: Assume p crashes. Then $\mathbf{T}_D$ is the random variable that represents the time that elapses from p's crash to the time when q starts suspecting p permanently. This time is closely related to the completeness property of the failure detector which requires such a suspicion *eventually*.

Mistake recurrence time $\mathbf{T}_{MR}$: Assuming a run without crashes, this random variable measures the time between two consecutive mistakes. A mistake occurs every time the failure detector was correct (i.e., no suspicion) and then erroneously starts suspecting another (correct) process.

Mistake duration $\mathbf{T}_M$: Again assuming a run without crashes, this random variable measures the time it takes the failure detector to correct a mistake.

Based on the latter two primary metrics, the following additional *derived metrics* are defined:

Good period duration $\mathbf{T}_G$: This random variable measures the length of a good period. For an ergodic probability distribution of failure detector histories, we have $\mathbf{T}_G = \mathbf{T}_{MR} - \mathbf{T}_M$.

Forward good period duration $\mathbf{T}_{FG}$: This is the random variable representing the time that elapses from a random time at which q trusts p to the next suspicion. The calculation of $\mathbf{T}_{FG}$ can be found in [CTA00].

Average mistake rate λ_M: This measures the rate at which a failure detector makes mistakes. For an ergodic probability distribution of failure detector histories, we have $\lambda_M = 1/E\{\mathbf{T}_{MR}\}$.

Query accuracy probability P_A: This is the probability that the failure detector's output is correct at a random time. For an ergodic probability distribution of failure detector histories, we have $P_A = E\{\mathbf{T}_G\}/E\{\mathbf{T}_{MR}\}$.

For non-ergodic failure detector histories, the relations between primary and derived metrics are more complex [Che00].

Since the system in this thesis and almost all other literature on failure detectors is not described by a probabilistic behavior but by bounds on stabilization time and transmission delays these parameters cannot be calculated. However, sometimes we can derive a bound on the detection time and/or the time until the failure detector stops making mistakes forever.

3.10 Implementation Principles

In general, there are two basic methods for implementing failure detectors. In the first approach the monitored process is responsible for sending messages and the monitoring process only listens. In the second case, the monitoring process queries a response from the monitored process.

Heartbeat based failure detectors. Here, the monitored process sends periodic messages ("heartbeats") to all monitoring processes. The monitoring process waits a certain time for the heartbeat and if not received, suspects the process. The waiting time is mostly determined by some local clock and a timeout value. The timeout value for the heartbeats can be calculated and adapted in various ways (e.g. [BMS02], cf. also Chapter 5). It is also possible to use order properties [BKM97] to detect a process that has stopped sending heartbeats.

Query based failure detectors. Such a failure detector uses a ping-pong principle in the sense that the monitoring process sends a request to the monitored process and waits for the answer of this process. If this answer is not received in time, the process is suspected. This approach is chosen either for efficiency reasons [LFA99] or for message-driven algorithms, where no local source of event generation exists (cf. Chapters 2 and 7).

3.11 Failure Detectors Implemented in this Thesis

We give a short overview of the failure detector implementations in the following chapters. We indicate either if a global (cf. Sections 3.3 and 3.4), local (cf. Section 3.6) or a transformation from a local to a global failure detector is presented in the respective chapters. Further the table shows the exact failure detector class and the implementation principle (cf. Section 3.10).

	Chapter 4	**Chapter 5**	**Chapter 6**	**Chapter 7**
type	global	global	local, transformation	local
class	Ω	$\diamond\mathcal{P}_p$	$\diamond\mathcal{P}_\ell$,$\diamond\mathcal{P}$	$\diamond\mathcal{P}_\ell$
impl. principle	heartbeat	heartbeat	heartbeat	query

Chapter 4

Weak Synchrony

Failure detectors should only be based on minimal assumptions on the amount of synchrony of the system in order to achieve high assumption coverage when implemented in real systems. When implementing a failure detector directly on a sparse network, it would be desirable if there is no need for more synchrony as in the fully connected case. In this chapter we provide evidence that indeed implementing a failure detector for a sparse network does not require stronger synchrony assumptions than implementing a failure detector for the fully connected case.

For this purpose we consider the model recently proposed by Aguilera et al. [ADGFT04]. They provided an implementation for Ω that works with very weak reliability and synchrony assumptions for links. In more detail, they need only f eventually timely links, to implement Ω, whereas all other links may be fair lossy. In such a model, as the authors show, it is not possible to implement $\diamond\mathcal{P}$. It is thus the first published model in which $\diamond\mathcal{S}$ resp. Ω is implementable but $\diamond\mathcal{P}$ is not.

In this chapter, we are generalizing their algorithm for sparse networks and show that in the sparse network case we

- need only the same synchrony assumptions as for the fully connected case,
- have no loss in fault-tolerance, and
- are even better in terms of message complexity.

4.1 Problem Specification

We describe the model of [ADGFT03b] in terms of our definitions in Chapter 2.

- Processes can fail by crashing, and the number of processes that may crash is bounded by the minimal degree of the network, i.e., $f < \delta$. We assume the com-

munication graph does not partition due to these failures. This can be ensured by choosing $f < \kappa$, where κ is the node connectivity[1] of the network graph.

- Links need not to be bidirectional. Links never crash and can be either fair lossy or *eventually timely* ($\diamond$-timely), i.e., we have t_{GST} and τ^+ finite but unknown. We call a node with j $\diamond$-timely outgoing links a $\diamond j$-source. We assume there is at least one correct $\diamond f$-source in the system.

- The algorithms are timed, i.e., we need local clocks with neglectable drift.

4.2 Related Work

The original algorithm of [ADGFT03b] is shown in Figure 4.1. It works in a fully connected network with at least one $\diamond f$-source while all other links may be fair lossy. Intuitively the algorithm works as follows: Every process p keeps a non-decreasing variable $counter_p[q]$ for every other process q in the system. Process p chooses as its leader the process q with smallest $counter_p[q]$, breaking ties using the process id. A counter $counter_p[q]$ is increased every time $n - f$ processes report that they suspect q. Every process periodically sends an (ALIVE, *counter*) message to all other processes, which itself use an adaptive timeout to build an estimate of all crashed processes. If the timeout at p for a process q runs out, p sends a (SUSPECT, q) message to all other processes. The key idea is that at every process p, for every $\diamond f$-source q, $counter_p[q]$ eventually stops increasing, while for all faulty processes this counter increases forever. Thus eventually a unique leader can be elected.

Variants of the same (weak) model together with appropriate algorithms appear in [ADGFT03a] and [ADGFT04]. With higher synchrony and reliability assumptions (e.g. there is a process with timely links to all other processes) also communication efficient implementations are possible.

4.3 The Sparse Network Algorithm

The algorithm presented in Figure 4.2 has been derived from the fully connected one of [ADGFT03b]. In fact, there are only a few changes to reflect the nature of sparse networks:

- In contrast to the original algorithm, the COUNTER and the ALIVE message are sent separately. The rationale behind this is that only constant size ALIVE messages have to be $\diamond$-timely, whereas the counter message, which is of unbounded size, needs not

[1] A graph G is κ-*node connected* if the removal of any set of $\kappa - 1$ nodes leaves the graph connected, whereas there is a set of κ nodes whose removal partitions the graph in at least two components.

```
1  initially
2      ∀q ≠ p : Timeout[q] ← T + 1
3      ∀q : counter[q] ← 0
4      ∀q : suspect[q] ← 0
5      ∀q ≠ p : reset timer(q) to Timeout[q]
6
7  repeat forever
8      leader ← ℓ such that (counter[ℓ], ℓ) = min{(counter[q], q) : q ∈ Π}
9
10 repeat forever
11     every T time steps do:
12         send (ALIVE, counter) to all processes except p
13
14 upon receive (ALIVE, c) from q do
15     for each r ∈ Π do counter[r] ← max{counter[r], c[r]}
16     reset timer(q) to Timeout[q]
17
18 upon expiration of timer(q) do
19     Timeout[q] ← Timeout[q] + 1
20     send (SUSPECT, q) to all
21     reset timer(q) to Timeout[q]
22
23 upon receive (SUSPECT, q) from r do
24     suspect[q] ← suspect[q] ∪ {r}
25     if |suspect[q]| > n − f then
26         suspect[q] ← ∅
27         counter[q] ← counter[q] + 1
```

Figure 4.1: The original algorithm from Aguilera et al. Code for a process p. The constant T is an arbitrary time value.

```
1  initially
2      ∀q ∈ nb(p) : Timeout[q] ← T + 1
3      ∀q : counter[q] ← 0
4      ∀q, r : suspect[q][r] ← 0
5      ∀q : lastseq[q] ← 0
6      ∀q : sequencer[q] ← 0
7      ∀q ∈ nb(p) : reset timer(q) to Timeout[q]
8
9  repeat forever
10     leader ← ℓ such that (counter[ℓ], ℓ) = min{(counter[q], q) : q ∈ Π}
11
12 repeat forever
13     every T time steps do:
14         send (ALIVE) to all neighbors
15         send (COUNTER, counter) to all neighbors
16
17 upon receive (ALIVE) from q do
18     reset timer(q) to Timeout[q]
19
20 upon receive (COUNTER, c) from q do
21     for each r ∈ Π do counter[r] ← max{counter[r], c[r]}
22
23 upon expiration of timer(q) do
24     Timeout[q] ← Timeout[q] + 1
25     sequencer[q] ← sequencer[q] + 1
26     send (SUSPECT, q, p, sequencer) to all neighbors and p
27     reset timer(q) to Timeout[q]
28
29 upon receive (SUSPECT, q, r, seq) do
30     if suspect[q][r] < seq then
31         suspect[q][r] ← seq
32         send (SUSPECT, q, r, seq) to all neighbors
33         if |{r|suspect[q][r] > lastseq[q]}| > deg(q) − f then
34             lastseq[q] ← max_s(suspect[q][s]) + 1
35             counter[q] ← counter[q] + 1
```

Figure 4.2: The modified version of the algorithm from Figure 4.1. Code for a process p.

to be timely. We believe that it is more realistic to timely transmit a constant size message between neighbors than a message of arbitrary size to any process in the system, if such messages are equipped with a higher priority.

- SUSPECT messages are forwarded to non-neighbors. Each suspect message carries a sequence number and the identifier of the origin, so that each suspect is counted only once, even if it received multiple times by a process.

- Since ALIVE messages are exchanged only with neighbors, the quorum for increasing $counter[q]$ is $\deg(q)-f$ (instead of $n-f$ as in [ADGFT03b]). If not a priori available, the information about the node degrees of the system can easily be distributed by an asynchronous algorithm.

The proof of correctness closely follows the original one, but has to consider the changes of the algorithm due to the sparse topology:

Lemma 4.1. *If p is a correct $\diamond f$-source, then for every q, $counter_q[p]$ is bounded.*

Proof. Since p is correct there are f neighbors of p that eventually do not timeout on p and therefore stop sending $(\text{SUSPECT}, p, \_, seq)$ messages. This implies that there is a seq_{max} such that $sequencer[p] < seq_{max}$ for f neighbors of p. Therefore no process will receive more than $\deg(p)-f$ SUSPECT messages with unbounded sequencer and therefore at every process the condition in line 33 becomes true only a finite number of times. Thus also by line 10, at process q the counter $counter_q[p]$ is bounded. □

Lemma 4.2. *If p and r are correct processes then for every time t and every process q, there exists a time after which $counter_r[q] \geq counter_p[q](t)$.*

Proof. Let $x = counter_p[q](t)$. If $x = 0$ or $p = r$ then the lemma holds because $counter_p[q]$ is nondecreasing. So assume $x > 0$ and $p \neq r$. After p sets $counter_p[q] = x$, it sends infinitely many COUNTER messages to its neighbors, where $counter[q]$ of these messages is at least x. Since links are fair lossy, eventually every neighbor receives one of these messages and sets its counter to a value greater or equal to x. By an inductive argument, every correct process in the network finally has $counter[q] \geq x$. □

Lemma 4.3. *If p is correct and q is faulty, then $counter_p[q]$ is unbounded.*

Proof. If q is faulty, at least $\deg(q)-f+1$ neighbors of q will suspect q forever and will send infinitely many SUSPECT messages with ever increasing sequence counters. Therefore, at every correct process p, the condition in lines 30 and 33 will become true infinitely often and therefore $counter_p[q]$ will increase infinitely often. □

Theorem 4.1. *The algorithm in Figure 4.2 implements Ω in a non-partitionable sparse network with fair lossy links and at least one correct $\diamond f$-source.*

Proof. By Lemma 4.1 and Lemma 4.3, and the fact that there is a correct $\diamond f$-source, it follows that for every correct process p, there is a time after which $leader_p$ is correct and stops changing. By Lemma 4.2, for every correct processes p and q, there is a time after which $leader_p = leader_q$. □

4.4 Complexity Analysis

In this section we compare the message complexity of the sparse network algorithm with the original one. The original algorithm assumes a fully connected network, which has in practice to be simulated atop of the sparse network. A message in the fully connected network has thus to be routed over multiple hops in the underlying sparse network. Since we have no information about the overhead due to routing we assume perfect routing along the shortest path. Typically, Δ is a constant for sparse networks, i.e., $O(\Delta) = O(1)$, which implies $O(d) = O(n)$. Note that that the separation of ALIVE and COUNTER messages is not relevant for the comparison, thus we consider them as a single message in both cases.

First consider the ALIVE messages. In both algorithms, they are broadcast every T time steps. However, in the original algorithm they are sent to every other process in the system, resulting in $n(n-1)$ messages in T time steps. On the other hand, a process running the sparse network algorithm sends the ALIVE messages to neighbors only, resulting in an overall message load of $\sum_{p\in\Pi} \deg(p) = |\Lambda| \leq n\Delta \leq n(n-1)$. Since in a sparse network we often have $\Delta << n$, there is a substantial reduction of message load. Moreover, the original algorithm has to send messages over several hops whereas in our algorithm all messages are sent between neighbors. Assuming an average distance of $d/2$ and perfect routing for the fully connected case, we have $|\Lambda|$ versus $n(n-1)d/2$ messages.

The message complexity of the SUSPECT messages is more difficult to calculate, since some timeouts might be increasing forever while others stop increasing. For simplicity of analysis, we assume all processes increase the timeouts forever and at the same time, and we calculate the message load for one such timeout interval, independent of its size. Since these simplifications affect both algorithms in the same way, this still allows a good comparison of the message complexity.

Once a process timeouts on another process, the original algorithm sends a suspect message to all other $n-1$ processes. Since every process monitors $n-1$ other processes we have $n(n-1)^2$ suspect messages that are routed independently over the network in the worst case. In contrast, the sparse network algorithm sends only $\sum_1^n \deg(p) \cdot |\Lambda| = |\Lambda|^2$ messages. Again, assuming an average distance of $d/2$ and a perfect routing for the fully connected case, we have $|\Lambda|^2$ versus $n(n-1)^2 d/2$ messages.

Putting everything together, in the low level sparse network, the original algorithm requires $O(n^3)$ ALIVE messages in T time steps and $O(n^4)$ SUSPECT messages in a timeout interval. The sparse network algorithm requires only $O(n)$ ALIVE messages and $O(n^2)$

suspect messages in a timeout interval. This does still not include the fact that employing a perfect routing in a fault prone network is very hard and thus additional overhead has to be expected for the original algorithm.

We are aware that the authors of the original algorithm already stated that the simple algorithm is inefficient and provided an efficient solution, where only f links of the fully-connected network carry messages forever. But this algorithm requires reliable links, which is not as weak as the model we are considering here.

4.5 Discussion

At a first glance, the Ω-implementation for sparse graphs seems to be the same as for the fully connected case. After all, we just made the communication with non-neighbors explicit, instead of simulating a fully connected point-to-point network. However, there are some more intricate differences.

The complexity analysis showed a significant reduction in message load in sparse networks, when we consider a sparse network model for the algorithm, instead of simulating a fully connected network. One may argue that this impairs fault-tolerance and hence the achievable assumption coverage. However, if a neighbor is not timely, the messages of a non-neighbor routed over this process must in general be considered non-timely as well. Therefore, restricting the f $\diamond$-timely links to links between neighbors is no loss of assumption coverage. Assuming $\delta > f$ does not reduce the assumption coverage either, since if this requirement was violated, the fully connected overlay graph could also partition and therefore violate the system model. Moreover, a bound on the transmission delays of constant size messages between neighbors (without routing layer) can be established with very high coverage. This yields an increase of assumption coverage compared to the original solution.

Chapter 5

Localized Services

In this chapter we present an implementation of a failure detector for a network, where processes are connected by a partitionable, sparse network with bounded number of neighbors. In wireless ad-hoc networks, where hardware limitations (e.g., receiver channels or buffers) allow processes to keep connections only to a certain number of other processes, this is a natural property of the underlying network. The total number of processes needs not to be known. Links and processes can fail by crashing, and due to these failures the network may partition into components. Processes can communicate only with their neighbors via a local broadcast primitive, which can also be implemented efficiently in wireless ad-hoc networks.

Processes do not need to know a bound on the communication delay between arbitrary processes but only a bound on the jitter of the communication between neighbors. This implies synchronous communication and may appear to be a severe restriction. Synchrony is required only between direct neighbors, however, which makes communication with non-neighbors at least partially synchronous [DLS88, LFA02] in case of unknown network size. In fact, in a wireless ad-hoc network, bounded communication delays can easily be achieved if a fixed part of the communication bandwidth is reserved for the failure detector, and the number of neighbors and the message size can be bounded. The algorithm of this chapter fulfills these conditions.

The algorithm uses heartbeats and timeouts to determine whether there is a connection between two processes. In contrast to systems where a fully connected network is assumed (and therefore the information is routed over the partially connected network when simulating the fully connected one), every process reuses the information of its neighbors, so unnecessary traffic can be avoided: Periodically, every process increments its own heartbeat counter and exchanges heartbeats with its neighbors. Consider a simple algorithm first, where every process forwards the heartbeats from itself and all other processes in the system to its neighbors in every round. Every process receives a new heartbeat from every connected process in each round here. If a process does not receive a heartbeat from another process, it suspects it. Still, this algorithm would require every

process to send $O(n)$ messages in every round.

By contrast, our algorithm forwards heartbeats of processes that are far away less frequently than those of nearer ones. The failure detector provides therefore more accurate information about nearer processes. This blends nicely with real systems, where, due to *localization*, processes that work together are often situated in the same region of the network. When choosing the parameters appropriately, we can reduce the traffic of each process so that every process sends in a time interval only a constant number of messages of logarithmic size, and yet calculate a precise timeout value.

5.1 Problem Specification

To achieve the network properties described above, we specialize the system model of Chapter 2. We assume persistent crashes of processes and links only. Further, we assume that there is a bound Δ on $\Delta(t)$ that is known to the processes. All links are bidirectional and reliable.

Processes can communicate with their neighbors using a *local broadcast* service. Such a service can be easily built from the send and receive primitives of the message passing model. In many architectures, however, such a primitive can naturally arise, if communication takes place via a shared medium of bounded extent. Consider, e.g., a wireless network, where all nodes in the communication range might be able to listen to a message. The service consists of two primitives, broadcast and deliver. When a process p invokes broadcast(msg) at time t, then deliver(msg) is triggered in the interval $[t+\tau^-, t+\tau^+]$ at all processes that are in $\mathrm{nb}(p, t+\tau^+)$. All links are therefore reliable until they crash. Note that a bound on the jitter $\varepsilon = \tau^+ - \tau^-$ needs to be known by the processes, which implies a synchronous communication.

We implement an eventually perfect failure detector $\diamond\mathcal{P}_p$ for partitionable systems that fulfills the properties strong completeness and eventual strong accuracy (cf. Section 3.5).

5.2 Related Work

Failure detectors for partitionable systems were defined in a similar way as in Section 3.5 by various authors [ACT99, BDM01, DFKM97]. In [ACT99], a sparsely connected communication graph is used, but the failure detector named "heartbeat" is used for quiescent reliable communication and is weaker than $\diamond\mathcal{W}$.

Many failure detector implementations of $\diamond\mathcal{P}$ use heartbeats [GCG01, BMS02, BMS03]. To our knowledge, none of them operates in sparsely connected networks, however. They could be used in conjunction with routing or other mechanisms to implement a reliable point-to-point network on top of a sparse network, but this typically creates excessive traffic.

To reduce communication traffic, methods like gossiping [vRMH98, WK03] are used, but these algorithms provide no deterministic solution and also require a fully connected communication graph.

Efficient failure detector implementations are given by Larrea et al. [LFA99]. Their efficieny criteria differs from ours since they consider an algorithm to be efficient if eventually the algorithm keeps sending messages over only $O(n)$ links. However, the number of messages on these links is not bounded. In constrast, our algorithm *always* sends only $O(1)$ messages of size $O(\log n + \log t)$ in a time inverval of T on every sparse network link. We believe this is a more interesting property, since this allows the implementation of links with known timing properties, as required by our algorithm.

5.3 The Algorithm

Every process p has, for every other process q it knows of, a heartbeat table consisting of:

- a heartbeat counter $hbc_p[q]$ which contains the most recent heartbeat of q
- a distance counter $distance_p[q]$ that contains p's estimate about the current distance to q
- a time-stamp $last_p[q]$ that holds the last round when p received a new heartbeat from q.

These arrays grow dynamically with each new process p learns of. For reasons of simplicity they are used in the algorithm as if they were statically allocated. The set $trusted_p$ contains all processes that the failure detector does not suspect, i.e., the failure detector's output $\tilde{H}(p,t)$.

The algorithm for process p is presented in Figure 5.1. It comprises a periodical task, which sends some part of its local heartbeat table to its current neighbors, and a receiver task, which updates the local table when it receives new heartbeats from neighbors.

Initially, p knows and trusts only itself. Every T time steps, p increases its own heartbeat counter, which is also used as the local round number. Every Δ^k rounds, all known processes with distance k are put into the set $unsent_p$. Since only messages from this set are sent, this ensures that every heartbeat of a process with distance k is broadcast at most every Δ^k rounds. From this set, the process id, heartbeat, and distance of the $\Delta + 1$ processes with lowest distance to p are sent and removed from $unsent_p$ in every round. If p does not receive a new update from another process it has previously trusted for a sufficiently long time, it suspects it.

The receiver task of p increases the distance counter of each message it receives by one. If this distance counter is shorter than its own estimate, it adopts the new distance. Once it receives a heartbeat newer than its own, it adopts this heartbeat and—if this is not already the case—trusts the heartbeat's origin.

```
1  variables
2      ∀q ∈ Π :  hbc_p[q], distance_p[q], last_p[q] ∈ ℕ        /* heartbeat table */
3      unsent_p ⊆ Π
4      trusted_p ⊆ Π                                          /* failure detector output */
5
6  initially
7      ∀q :  hbc_q[p] = 0
8      distance_p[p] = 0, ∀q ≠ p :  distance_p[q] = ∞
9      trusted_p = {p}
10     unsent_p = ∅
11
12 every T time steps do:
13     hbc_p[p] = last_p[p] = hbc_p[p] + 1                    /* the local round number */
14     for each k ≥ 0, such that Δ^k divides hbc_p[p] do:
15         add all q with distance_p[q] = k to unsent_p
16     for 1 to Δ + 1 do:
17         q = an item from unsent_p for which distance_p[q] is minimal
18         remove q from unsent_p
19         broadcast_p(q, hbc_p[q], distance_p[q])
20     for each q ∈ trusted_p do:
21         if (hbc_p[p] − last_p[q])T > Timeout(distance_p[q]) then
22             remove q from trusted_p                        /* suspect q */
23             distance_p[q] = ∞
24
25 on deliver_p(q, new_hbc, new_dist) do:
26     if  distance_p[q] > new_dist + 1 then
27         distance_p[q] = new_dist + 1
28     if  new_hbc > hbc_p[q] then                            /* more recent heartbeat */
29         hbc_p[q] = new_hbc                                 /* adopt heartbeat */
30         last_p[q] = hbc_p[p]                               /* set reception timestamp */
31         if q ∉ trusted_p then
32             add q to trusted_p                             /* trust q */
33
34 function Timeout(k) = (2T/(Δ−1))Δ^k + kε
```

Figure 5.1: Failure detector algorithm for any process p. It comprises a periodical task and a message handler.

5.4 Proof of Correctness

Theorem 5.1. *The algorithm in Figure 5.1 implements strong completeness.*

Proof. If p and q become disconnected, eventually $hbc_p[q]$ will not grow anymore, since only q can increase $hbc_q[q]$, and only connected processes can learn this value. Either p already suspects q, in which case we are done, or $distance_p[q]$ is bounded by the last distance between p and q. Thus, eventually p will time out q and therefore suspect it. □

To show eventual strong accuracy, we need some technical lemmata. First, we derive a bound on the number of processes that are at a certain distance to a process:

Lemma 5.1. *Let p be any process in a network with maximal degree Δ. Then $n_0(p,t) = 1$ and for $k > 0$, $n_k(p,t) \leq \Delta(\Delta-1)^{k-1}$.*

Proof. Obviously, p is the only process with distance 0 to p. For $k = 1$, since p can have at most Δ neighbors, the lemma also holds. Assume that the lemma is valid for $k-1 > 0$. Then $n_{k-1}(p,t) \leq \Delta(\Delta-1)^{k-2}$. Each of these processes must have a link to some process in P_{k-2}. Therefore at most $\Delta - 1$ links can lead to processes in P_k, yielding $n_k = |P_k| \leq \Delta(\Delta-1)^{k-1}$ □

The variable $distance_p[q]$ at time t is p's estimate of $D(p,q,t)$. If a longer path is faster than a shorter one, this estimate may not be exact, but it is never smaller than the initial real distance:

Lemma 5.2. *For any $q \in P_k(p,0)$, $distance_p[q] \geq k$.*

Proof. The variable $distance_p[q]$ is only set when p receives a heartbeat of q. At q, $distance_q[q] = 0$, and with every hop along the path from q to p this hop counter is increased. Therefore $distance_p[q]$ contains the length of the path the heartbeat took. By definition every process in $P_k(p,0)$ is initially at distance k to p, and the distance is monotonically increasing. Therefore the length of this path must be greater or equal than k. □

Lemma 5.3. *A heartbeat of a process $q \in P_k(p,0)$ is sent by p at most once every Δ^k rounds.*

Proof. A process q's heartbeat is sent by p only if it is previously put into $unsent_p$. This happens only every $\Delta^{distance_p[q]}$ rounds (lines 14+15). According to Lemma 5.2, $distance_p[q] \geq k$ and therefore $\Delta^{distance_p[q]} \geq \Delta^k$. □

The following lemma shows that, although our algorithm sends constant size messages at every process, our scheduling function ensures that the heartbeat counter of every process is forwarded periodically by every process. In the following, we will call the value of $hbc_p[p]$ also the current round number. Note that no global rounds exist, round numbers are only local.

Lemma 5.4. *If $k = distance_p[q]$ at some time t, the heartbeat of q is broadcast by p at most $2\Delta^k$ rounds after t.*

Proof. Let $Q = \{q' | distance_p[q'] \leq k\}$ the set of processes with an estimated distance to p less or equal than q. We first show that the maximum number m of messages containing heartbeats from processes in Q and sent in Δ^k rounds is less or equal to $\Delta^k(\Delta + 1)$. Let q' be a process from Q and let $i = D(p, q', 0)$ be the initial distance of p and q' (that is, $q' \in P_i(p, 0)$). According to Lemma 5.3, q' causes at most Δ^k/Δ^i messages in Δ^k rounds. Therefore, the sum m of all messages from Q is less or equal than

$$\sum_{i=0}^{k} n_i(p, 0)\frac{\Delta^k}{\Delta^i},$$

and using Lemma 5.1,

$$m \leq \Delta^k \left(1 + \sum_{i=1}^{k} \frac{\Delta(\Delta - 1)^{i-1}}{\Delta^i}\right) \leq \Delta^k(\Delta + 1)$$

Since $\Delta^k(\Delta + 1)$ messages can be sent in Δ^k rounds according to lines 16-19, and since less or equal than $\Delta^k(\Delta + 1)$ heartbeats (including q) can have a higher or equal priority than q, q is broadcast in each period of Δ^k rounds at least once.[1] Since the position of t in the Δ^k period can be arbitrary, the heartbeat is broadcast at most after $2\Delta^k$ rounds. □

The guarantee that a heartbeat is forwarded after some well defined time allows us to compute a precise bound on the time after which a process learns the current round number of another process in the system:

Lemma 5.5. *If two processes p and q remain connected with distance k after some time t_0 and p increases $hbc_p[p]$ at time $t \geq t_0$ to a value v, then q sets $hbc_q[p] = v$ and $distance_q[p] \leq k$ by time $t + k\tau^+ + 2(\sum_{i=1}^{k-1} \Delta^i)T$.*

Proof. By induction on k. For $k = 1$, p broadcasts p immediately and therefore q receives $hbc_p[p]$ at least after τ^+ time steps. For $k > 1$, assume the lemma holds for $k - 1$. Let q' be a neighbor of q with distance $k - 1$ from p (since $\forall t' \geq t : D(p, q, t') = k$, such a process exists). Then by the induction hypothesis, q' sets $hbc_{q'}[p] = v$ and $distance_{q'}[p] \leq k - 1$ by time $t + (k - 1)\tau^+ + 2(\sum_{i=1}^{k-2} \Delta^i)T$. Therefore, according to Lemma 5.4, q' forwards the heartbeat and distance of p at most after $2\Delta^{distance_{q'}[p]}T \leq 2\Delta^{k-1}T$ time steps. In consequence, including the maximum communication delay τ^+, q receives v at $t + k\tau^+ + 2(\sum_{i=1}^{k-1} \Delta^i)T$. According to line 26+27 of the algorithm, after the reception of this message, $distance_q[p] \leq k$. □

[1] In fact, q it is broadcast *exactly* once in Δ^k rounds.

With that result, we can compute a timeout on the time difference between two updates of a heartbeat counter. This timeout does not depend on any other parameter than the maximum degree Δ and the jitter ε:

Lemma 5.6. *If any two processes p and q remain connected at distance k, the time difference between two updates of $hbc_q[p]$ at q is less or equal than $\mu\Delta^k + k\varepsilon$, where $\mu = \frac{2T}{\Delta-1}$.*

Proof. W.L.O.G., assume that p sets $hbc_p[p] = v$ at time 0 and $hbc_p[p] = v+1$ at time T. Obviously, the earliest time q can set $hbc_q[p] = v$ is $k\tau^-$. By Lemma 5.5, the latest time q can set $hbc_q[p] = v+1$ (or a higher value) is $T + k\tau^+ + 2(\sum_{i=1}^{k-1} \Delta^i)T$. Hence, the time difference is

$$2\sum_{i=1}^{k-1} \Delta^i \cdot T + T + k\varepsilon \leq T \cdot \left(2\frac{\Delta^k - 1}{\Delta - 1} - 1\right) + k\varepsilon \leq \mu\Delta^k + k\varepsilon$$

□

With these lemmata we can prove the second failure detector property:

Theorem 5.2. *The algorithm in Figure 5.1 implements eventual strong accuracy.*

Proof. Let p and q be two processes that remain connected. After some time, $D(p, q, t)$ does not change anymore. Then either p never suspects q—in which case we are done—or there is a time where p suspects q. In this case, $distance_p[q] = \infty$. Then eventually p will learn k, i.e., $distance_p[q] = k$. According to Lemma 5.6, p receives a new heartbeat from q at least every $\mu\Delta^k + k\varepsilon$ time steps. Therefore the condition in line 21 is never satisfied and the processes never suspect each other. □

Corollary 5.1. *The algorithm in Figure 5.1 implements an* eventually perfect failure detector $\diamond\mathcal{P}_p$ *for partitionable systems.*

5.5 Complexity Analysis

In this section we analyze the message complexity and the failure detection time of the algorithm. As the following theorems show, we get the logarithmic message complexity in exchange for a failure detection time exponential in the distance between the nodes.

Theorem 5.3. *In every round of T time steps, every process*

- *sends at most $\Delta + 1$ messages*
- *receives at most $\Delta(\Delta + 1)$ messages*

where each message is of size $O(\log n + \log t)$.

Proof. That the number of messages a process sends per round is $\Delta+1$ follows immediately from lines 16 and 19 of the algorithm. Every node has at most Δ neighbors, so the number of received messages is $\Delta(\Delta+1)$. The message size follows from the fact that every message is a tuple $(p, hbc, distance)$, where p is of size $O(\log n)$, hbc of size $O(\log t)$ and distance of size $O(\log n)$. □

In practice, the message size can be regarded as constant. Since Δ is also a constant, the communication traffic at each process is of constant size.

Finally, we compute the time until two processes that become disconnected suspect each other. In [CTA00] (cf. Section 3.9), this is called the *failure detection time* T_D.

Theorem 5.4. *If two processes p and q become disconnected at time t, they suspect each other by time $t + 2\mu\Delta^k + k(2\tau^+ - \tau^-)$, where k is the distance just before the partition.*

Proof. According to Lemma 5.5, p receives the last heartbeat of q by time $t + k\tau^+ + 2(\sum_{i=1}^{k-1} \Delta^i)T$. According to line 21, p suspects q exactly $\mu\Delta^k + k\varepsilon$ time steps later. Since

$$t + k\tau^+ + T \cdot \left(2\frac{\Delta^k - 1}{\Delta - 1} - 1\right) + \mu\Delta^k + k\varepsilon \leq t + 2\mu\Delta^k + k(2\tau^+ - \tau^-)$$

the theorem follows. □

5.6 Summary and Discussion

We presented an implementation of an eventually perfect failure detector for a partitionable network with sparse topology. Between neighbors, an upper bound on the communication jitter is assumed. The number of neighbors is assumed to be bounded by Δ, which is an adequate model for wireless ad-hoc networks. The algorithm requires neither a priori knowledge of the number of processes in the system nor an upper bound on the communication delay between arbitrary processes. Every process broadcasts just $\Delta + 1$ messages per round to its neighbors, and under the assumption of a constant size name-space and time domain, these messages are of constant size. Processes at shorter distances get more accurate information about each other than farther ones.

It is possible to adapt our algorithm to systems where links can also recover. However, in such a system the definition of reachability is not obvious, since an application of the failure detector may use e.g. a routing algorithm for communication. The application-level reachability relation would hence also depend on the behavior of this routing algorithm.

Chapter 6

Fault-tolerant Self-stabilization

As many applications rely upon failure detectors, failure detector implementations should keep providing their service even in adversarial operating environments. We aim at two approaches here: Firstly, we weaken the system timing models as far as possible. Much recent work focuses on this topic—see [ADGFT03a] and [LLS03, Wid03, SW05]. We addressed this topic already in Chapter 4. And secondly, improve the availability of the failure detector by referring to the self-stabilization paradigm [Dij74]. This requires algorithms that stabilize (and remain in legitimate states) even in the presence of permanent faults [BKM97, Gär02, AH93, DDP03].

In this chapter we consider self-stabilizing implementations of failure detectors using time-driven approaches. Message-driven self-stabilizing failure detector algorithms are addressed in Chapter 7.

6.1 System Model

Self-stabilization requires an algorithm to recover from any (invalid) state in finite time. More formally, a self-stabilizing algorithm has to converge to a set of states—called legitimate states—from any state (convergence) and then has to remain in legitimate states (closure) [AG93]. We assume that our system stabilizes at some unknown time t_{GST}, after which the timing assumptions hold, the number of permanent faults is bounded and no further state corruption at correct processes occurs. On the other hand, at t_{GST}, processes may be in an arbitrary state and arbitrary messages may be in transit.

Before t_{GST}, arbitrarily changes of the network topology are possible, after that, only link and process crashes are allowed. In this chapter we consider time-driven solutions, i.e., the processes are equipped with local clocks that contribute to the progress of the algorithm. We drop this assumption in Chapter 7.

Due to faults, the network may partition into several components. Thus we aim for the failure detector classes $\diamond\mathcal{P}_\ell$ and $\diamond\mathcal{P}_p$.

6.2 Self-stabilization and Failure Detection

Like Tel [Tel94] we distinguish two approaches for how to deal with faults: Self-stabilization and robustness. Robust algorithms provide their functionality even in the presence of faults. However, the severity and number of faults usually have to be limited in order to guarantee the required properties. Self-stabilization, on the other hand, is able to recover from an arbitrary system wide error state in finite time. However, the desired services are only ensured during stable periods, where no new state corruption occur. Failure detectors are modules typical for robust algorithms, thus a self-stabilizing failure detector combines these two approaches to fault-tolerance.

When trying to formally define self-stabilization for failure detector implementations, a limitation in the original definition of self-stabilization become obvious: Dijkstra [Dij74] defined self-stabilization as follows: Starting from an arbitrary state, the algorithm always converges within finite time into a *legal state* and remains in legal states forever. However, the properties of a failure detector are not defined for states: Even if we consider a perpetual failure detector, which provides perpetual accuracy, every implementable completeness property can only require that a crash is detected *eventually*, resp., within bounded time. Consider a failure detector output that is accurate and complete at time t. Later, at time $t' > t$ some process crashes. At t' the failure detector output cannot be complete. It follows that the definition of legitimate states via the failure detector output would inevitably lead to a violation of closure. The correct behavior is thus not given by the current failure detector output, but on the whole execution of the failure detector. Nevertheless, it is possible to define the legal states for a failure detector by looking at all possible executions applicable to a distinct configuration. Since every failure detector that has stabilized will not suffer from any further failures, the set of all applicable and timely executions define the set of possible executions from that configurations. We thus have a definition that is equivalent to the one in [Tel94]:

Definition 6.1 (Legitimate configuration). *Let $\mathcal{A}$ be an algorithm that implements a problem defined by properties $\mathcal{P}_1(\sigma) \ldots \mathcal{P}_k(\sigma)$ on the execution σ of $\mathcal{A}$. Then a configuration C is a* legitimate configuration*, if every timely execution σ' that is applicable to C satisfies $\mathcal{P}_1(\sigma') \wedge \ldots \wedge \mathcal{P}_k(\sigma')$.*

Definition 6.2 (Self-stabilizing Algorithm). *We say an algorithm $\mathcal{A}$ of a problem $(\mathcal{P}_1, \ldots, \mathcal{P}_k)$ is* self-stabilizing*, if in every execution a legitimate configuration is reached after finite time, disregarding the initial state.*

When looking at an implementation of an eventually perfect failure detector $\diamond\mathcal{P}$, we observe at t_{GST} that neither strong completeness nor eventual strong accuracy is violated since they consist of requirements that have to hold only eventually. Thus from a formal viewpoint, every correct algorithm (in particular a self-stabilizing one) will fulfill the completeness property and the accuracy property at t_{GST}, resulting in a stabilization

time of zero. However, from an intuitive point of view, the algorithm has not stabilized. Thus, looking at a perpetual perfect failure detector $\mathcal{P}$ is more intuitive, since now the failure detector has stabilized if no further false suspicions will be made.

Now, from a formal viewpoint, the legitimate states of our algorithms are defined by the properties of a perpetual perfect local failure detector $\mathcal{P}_\ell$. It is obvious, however, that any self-stabilizing implementation of $\mathcal{P}_\ell$ can only be used as $\diamond\mathcal{P}_\ell$ by upper layer applications. Thus we will show for our algorithms the properties of $\diamond\mathcal{P}_\ell$.

6.3 The Need for Bounded Memory

Many algorithms in distributed computing assume unbounded local memory. In most cases, this requirement stems from the fact that such an algorithm uses a variable that may grow arbitrarily. For non-stabilizing systems, this is not really a problem, since in most cases, such a variable can be safely approximated by a sufficiently large bounded memory variable.

This is not reasonable for self-stabilizing systems. Consider e.g. an integer variable that is approximated by a fixed size integer in a real system. Since we require self-stabilization we have to deal with the case where this variable stabilizes just "before" the wrap-around. Thus, if there is no self-stabilizing bounded memory solution that solves a certain problem, a solution that uses approximated integers will never work. Therefore we require all our self-stabilizing algorithms to be bounded in the size of the local memory.

However, there are some cases where the approximation seems to work well: Assume an algorithm that estimates the a priori unknown number of nodes n in a system. Since n is unbounded, from a theoretical viewpoint it is not possible to have an algorithm that uses bounded local memory and that can store that information. By choosing a sufficiently large bound N on n we can cope with every real network of realistic size. However, in fact this algorithm assumes another network model, namely one with a known upper bound on the network size. Thus in the following we model this explicitly by assuming such a bound N.

The major impact of this bounded memory requirement is the fact that we cannot store the values for adaptive learning of unknown timeouts locally, thus all algorithms that use this technique do not work. Consequently, we do not know whether there is a bounded memory solution of even a weak failure detector in partial synchronous systems with unknown bounds. Here we have yet another problem when approximating the size of the domain of the timeout values: If we assume an implicit bound that will work in practice, the performance of such algorithms would be prohibitive, since an overly pessimistic timeout might emanate from the unstable period, while decreasing timeout values may violate failure detector properties.

```
1  state variables
2      suspect_p ⊆ nb(p)
3      ∀q ∈ nb_p : timeout_p[q] ∈ {0 ..., Ξ}
4
5  every T time steps do
6      send (ALIVE) to all neighbors
7      ∀q ∈ nb_p : timeout_p[q] ← timeout_p[q] ⊖ 1
8      suspect_p ← {q ∈ Π | timeout_p[q] = 0}
9
10 on receive (ALIVE) from some neighbor q
11     timeout_p[q] ← Ξ
```

Figure 6.1: A simple self-stabilizing algorithm implementing a self-stabilizing $\mathcal{P}_\ell$. The variable *suspect* contains the failure detector output

6.4 Simple Local Self-stabilizing Failure Detectors

We first provide a self-stabilizing solution for a local failure detector. This algorithm—shown in Figure 6.1 is really simple, in fact, it is presumably the most simple failure detector algorithm that is possible: Each process sends heartbeats every T time steps to each of its neighbors, each neighbor waits at most $\Xi \cdot T$ time steps before it suspects another process. Obviously, the algorithm works, if Ξ is a bound on ε/T. Except for the fact that these timing conditions need to hold only after t_{GST} (which matches one of the partially synchronous settings in [DLS88]), this model is very demanding in terms of synchrony.

However, weaker types of synchrony that consider the existence of *unknown* bounds, e.g. on the transmission delay [DLS88, CT96, ADGFT03b] or on the ratio between the minimal or maximal transmission delay conflict with the bounded memory requirement: All algorithms that we are aware of which use such a synchrony assumption require unbounded local memory and are thus not suitable for self-stabilizing systems.

We first prove the algorithm in Figure 6.1 to be correct:

Theorem 6.1. *The algorithm in Figure 6.1 is a self-stabilizing implementation of $\diamond\mathcal{P}_\ell$.*

Proof. We show the properties of an eventually local failure detector. Note that they hold independent of the initial state of the system.

Local Completeness: Assume process q crashes at time t. Let $t' = \max(t, t_{GST})$. Then every correct neighbor p does not receive any messages from q after $t' + \tau^+$, thus line `11` at process p is never executed for neighbor q after that. Hence, $timeout_p[q]$ is never increased anymore, but decreases by 1 every T time steps. Since $timeout_p[q]$ is at most Ξ, by time $t' + \tau^+ + \Xi T \leq t' + \tau^+ + \varepsilon = t' + 2\tau^+ - \tau^-$, q is suspected by p (line `8`) and remains suspected forever.

```
1   state variables
2       suspect_p ⊆ nb(p)
3       ∀r, s ∈ nb_p : count_p[r][s] ∈ {0 ..., m + 1}
4
5   every T time steps do
6       send (ALIVE) to all neighbors
7
8   on receive (ALIVE) from some neighbor q
9       suspect_p ← suspect_p − {q}
10      for all r ∈ nb − {q} do
11          if r ∉ suspect_p
12              count_p[q][r] ← count_p[q][r] + 1
13              if count_p[q][r] > m
14                  suspect_p ← suspect_p ∪ {r}
15          count_p[r][q] ← 0
```

Figure 6.2: The self-stabilizing failure detector of [BKM97].

Eventual Local Accuracy: If a neighbor q of p is correct, after t_{GST} it sends messages every T time steps. Thus by time $t_{GST} + T + \tau^+$, p receives a message from q, and sets $timeout_p[q] = \Xi$. After that, $timeout_p[q]$ will never become 0, because this would require $T\Xi = \varepsilon$ time, but by that time line `11` has already been executed again. So from time $t_{GST} + 2T + \tau^+$, q remains not suspected forever. □

A more relaxed synchrony assumption is described in [BKM97]: It assumes that there is a bound m on the *reception rate* of messages. This synchrony assumption is strictly weaker than the one considered above, if we assume only non-zero transmission delays. In particular it is time-free in the sense that the transmission delays need not satisfy any timing condition but only have to establish an order property. The algorithm for this system is shown in Figure 6.2. Note that in contrast to the Θ assumption (cf. Section 2.5.4) this model is time-driven. Furthermore, here the system model is not independent of the algorithm: The bound m can only hold if all processes send messages periodically at a given rate $1/T$.

6.5 Stable Failure Detector Transformation

In this section we provide a transformation from a local failure detector of class $\diamond\mathcal{P}_\ell$ to a global failure detector of class $\diamond\mathcal{P}_p$. The self-stabilizing transformation algorithm is asynchronous but time-driven and shown in Figure 6.3. In a nutshell, the algorithm uses flooding to distribute the local view of each process to every other process. Each process

```
1  input localsuspect_p ⊆ nb(p)
2  output globalsuspect_p ⊆ Π
3
4  state variables
5      ∀r, s ∈ Π :  netview_p[r][s] ∈ {true, false}        /* true if link (r, s) is trusted */
6
7  every T time steps do
8      ∀q ∈ Π :  netview_p[p][q] ← (q ∉ localsuspect_p)
9      send (p, localsuspect_p, N) to all neighbors
10
11 on receive (q, S, hopcounter)
12     ∀r ∈ Π :  netview_p[q][r] ← (r ∉ S)
13     if hopcounter > 0
14         send (q, S, hopcounter − 1) to all neighbors
15     globalsuspect_p = {q | there is no path from p to q in netview_p}
```

Figure 6.3: Self-stabilizing transformation from $\diamond\mathcal{P}_\ell$ to $\diamond\mathcal{P}$. N is a bound on n and T an arbitrary time value.

holds a matrix with the information of the local suspicions of all other processes, which provides an eventually correct view of all links to every process. A process p globally suspects another process q only if there is no path from p to q in p's matrix.

By not only exchanging process information but link information, the algorithm provides a nice stability property: $2N\tau^+ + T$ time after all local failure detectors have stabilized, i.e., stopped making any incorrect suspicion, also the global failure detector given by the composition of all local failure detectors and the transformation algorithm stops making mistakes.

Before going into more detail w.r.t. the stabilization time we show the correctness of the transformation algorithm:

Theorem 6.2. *The algorithm in Figure 6.3 is a self-stabilizing transformation from $\diamond\mathcal{P}_\ell$ to $\diamond\mathcal{P}$.*

Proof. We show the properties of an eventually perfect failure detector $\diamond\mathcal{P}$, under the assumption that $localsuspect_p$ fulfills the properties of $\diamond\mathcal{P}_\ell$. Note that the values of the state variables can be initially arbitrary.

Strong Completeness: Assume by contradiction that q crashes or becomes partitioned from p at time t but there is no time t' such that process p suspects q for all times after t'. This is only the case if there is a path from q to p in $netview_p$ for all times after some time t''. Let $\pi = p_1 \dots p_k$, with $q = p_1$ and $p = p_k$, be such a path that is in $netview_p$ for all times $t''' > t''$. Either all processes p_i for $1 \leq i < k$ and all links (p_i, p_{i+1}) are correct,

or there is at least one crashed process or link in this path. In the first case we are done, since this contradicts the assumption that p and q are not connected. In the second case, let p_j be the process in π with maximal j such that p_j is correct and either (p_{j-1}, p_j) or p_{j-1} has crashed. Then $localsuspect_{p_j}$ will eventually contain p_{j-1}. Since the path from p_j to p is correct by assumption, p will eventually learn that p_j locally suspects p_{j-1} and thus remove the link from $netview_p$. Contradiction to the fact that $netview_p$ contains π for all times after t''.

Eventual Strong Accuracy: Assume by contradiction that p and q never crash or get partitioned but p never stops suspecting q forever. A process q is suspected by p if there is no path q to p in $netview_p$. However, since links can only crash, after t_{GST} there is at least one permanent path $\pi = p_1 p_2 \dots p_k$ with $p_1 = q$ and $p_k = p$ in $G(t)$ from q to p, where all processes and all links on the path are correct. Thus, eventually every process p_{i+1} on this path will stop suspecting p_i forever. Further all messages from these processes will arrive at p, since the path is not longer than N and all links are correct. Since every process p_i now periodically broadcasts that it does not suspect p_{i+1}, eventually p has set $netview_p[p_i][p_{i+1}]$ to *true* for every link (p_i, p_{i+1}) in π forever. Contradiction. □

To see why the global failure detector stabilizes in $O(N\tau^+ + T)$ time after the local failure detectors have stabilized, consider the case where all local failure detectors stabilize at time t. After that no incorrect suspicion occurs, but messages that do not reflect this situation may be in transit. After $t + N\tau^+$, none of these messages can be in transit anymore due to the hopcounter. By time $t + N\tau^+ + T$, all processes broadcast their new view which is received by all connected processes by time $N\tau^+$ after that, which is at time $t + 2N\tau^+ + T$.

6.6 Summary

In this chapter we introduced the concept of self-stabilization and how it can be applied to failure detectors. We argued for the need of bounded memory algorithms and showed simple solutions to that problem. In contrast to the following chapter, we considered time-driven solutions. We further showed how the simple local failure detectors can be transformed into a global failure detector in a self-stabilizing way.

In the next chapter considers message-driven failure detector implementations, and we will see that under these circumstances the problem is not as easy as in the time-driven case.

Chapter 7

Message-driven Self-stabilizing Failure Detection

Generally, the discipline of distributed computing considers sets of distributed processes that execute algorithms where each execution consists of a sequence of events. In the context of reliable agreement problems, much work [CT96, DDS87, DLS88] focuses on timing constraints on these events; e.g. upper bounds between send and reception events of messages between processes (see Section 2.5). Another issue is event generation. We distinguish here two kinds of models, i.e., time-driven and message-driven. In time-driven algorithms, events occur due to passage of time and are triggered by clocks or timers. In contrast, when considering message-driven algorithms, after the algorithm was started, all events happen as immediate reaction to a received message while clocks are either not part of the model or are just not employed by the algorithms.

Note, however, that the issues of timing constraints and event generation are orthogonal. Consider, e.g., the well known failure detector based consensus algorithms of [CT96] which work in an asynchronous model of computation (often referred to as "time-free" model, reflecting the absence of timing bounds). These algorithms must be attributed as time-driven as steps can be taken—independently of the presence or absence of messages in input buffers—just by the passage of time respectively the progress of the program counter. It seems obvious that solutions to the same problem can be achieved with message-driven algorithms if

1. messages are immediately processed upon reception,
2. the failure detector module triggers the consensus algorithm if new suspicions have been added and
3. the failure detector implementation itself is message-driven.

Most existing failure detector implementation in the literature [CT96, ADGFT03a, BKM97] are not message-driven as they periodically send messages (e.g. heartbeats). Ex-

ceptions are the message-driven failure detector implementations of [LLS03, WLLS05] which show that failure detectors can be implemented without autonomous event generation (i.e., timers or clocks).

In this chapter, the problem of a message-driven self-stabilizing [Dij74, Dol00] implementation of failure detectors is investigated. The first self-stabilizing failure detector implementations were introduced by Beauquier and Kekkonen-Moneta [BKM97]. We showed them in Chapter 6. Their implementations send messages with every clock tick, in other words, they are time-driven. These algorithms satisfy the failure detector semantics and stabilize within finite time, i.e., they recover from an arbitrary state in systems that obey a fair ordering property, which is in fact an abstract synchrony assumption: If a process receives m messages from a process it must receive at least one message by any other correct process.

Obviously, self-stabilizing failure detector implementations cannot be purely message-driven as inaccurate failure detector information can never be corrected after a state was reached where no messages are in transit such that no process will ever make a step afterwards. To overcome such situations, several approaches can be taken. We could augment the system model with the requirement that at least one message must always be in transit. Since for the states where no messages are in transit convergence cannot be ensured we could not argue that recovery from *all* states is guaranteed. Moreover, when considering practical solutions this does not seem reasonable (e.g. this does not cover system booting [Wid03, WLLS05] where no messages are in transit initially). Therefore we add a local deadlock prevention event, which however has no timing constraints except that in every infinite run it happens an infinite number of times, where the duration between two events is finite. In other words this event cannot be used as a—even weak [FS04a]—clock. Strictly speaking, using this approach, our algorithms are not message-driven anymore. Our proofs, however, reveal that they do not rely on the deadlock prevention event if messages are in transit. In this chapter we show that under certain circumstances it is impossible to implement failure detectors in message-driven models even with such a deadlock prevention event. Trivially, this result also holds for purely message-driven systems. Therefore, having this deadlock prevention event in the system model makes the impossibility result even stronger as it holds not only for message-driven systems but also for message-driven systems with deadlock prevention.

Because of the considerations in Section 6.3, we focus on bounded memory failure detector algorithms under quite conservative synchrony assumptions, i.e., there exists some upper bound on message end-to-end delays that is different from the lower bound. For our implementations—but not for the impossibility result—we require the lower bound on message end-to-end delays to be greater than 0.

Under these assumptions it turns out that the number of messages that can be in transit at any given time becomes an important factor. Messages from the unstable period could produce message patterns which look identical to correct ones but which are much denser in the sense that the elapsed time of such a faulty pattern is just a fraction of the

duration of a corresponding correct pattern. We show in Section 7.3 that this behavior leads to the impossibility of implementing even the weakest failure detector [CHT96] that allows solving consensus, i.e., the eventually strong failure detector $\diamond\mathcal{S}$ (as defined in [CT96]) in fully connected networks if the number of messages that are simultaneously in transit is unknown.

By devising two failure detector implementations that work in sparse networks we show how to circumvent this impossibility result.[1] The first algorithm—discussed in Section 7.4—copes with an unbounded number of messages but requires unbounded space. Since we also want to give a practical solution, we devise a second algorithm in Section 7.5 which requires just bounded space. This algorithm, however, requires knowledge of M, an a priori upper bound on the number of messages that may be in transit simultaneously. Since real networks are finite, we consider this upper bound as not very restrictive. For many networks, M can be analytically derived as the capacity of links (determined by memory allocated to queues at the network layers) is bounded as well. However, this bound has no influence on the detection time of our algorithm and the space requirements are just logarithmic in M. So even in networks where it is difficult to find a tight upper bound, one can still use an extremely conservative one. The second approach is therefore of greater practical relevance.

Our impossibility result for implementing failure detectors is based on six assumptions: Self-stabilization, message-driven semantics, unknown bound on link capacity, bounded memory of the failure detector algorithm, determinism, and timing uncertainty. If any requirement is dropped the problem becomes solvable (cf. also Sections 7.6 and 7.7). This shows how much clocks/timers help when invalid message patterns must be tolerated. This is also the answer to the question whether time-driven and message-driven semantics are equivalent regarding expressiveness; in other words whether the same set of problems have solutions in both models. We answer the question in the negative.

7.1 System Model

We consider a sparse network with reliable, bidirectional links. Links and processes may fail by crashing. Due to this, the system may partition. However, we assume that every process has at least one correct neighbor.

All algorithms in this chapter are message-driven and self-stabilizing. The impossibility result is not restricted to any synchrony assumptions. For our algorithms we assume a bound Θ on the ratio between the minimum and maximum end-to-end communication delay (see Section 2.5, [LLS03]).

[1] Obviously, these algorithms work in fully connected networks as well.

7.2 On Deadlock Prevention Events

To our knowledge, there are three types of deadlock prevention mechanisms for message-driven algorithms:

1. the assumption that there is always at least one message in transit [DIM97],
2. the existence of an abstract deadlock detection mechanism [DIM97], which reacts either on local or global deadlocks; and
3. the use of a deadlock prevention event mechanism, that unlocks the algorithm from time to time.

The first possibility is no good choice for algorithms: They would not work if during the instable period all messages in the system get lost. However, for our impossibility result we consider an alive execution, which only strengthens our result. For our algorithms we assume the existence of a deadlock prevention event generation mechanism. In this section we formalize the latter two deadlock prevention mechanisms in an abstract module, and show that they can be transformed into each other.

A *deadlock detector* is a module located at each process, which outputs events called deadlock prevention events (DPE). The deadlock detector fulfills one of the following properties:

Global Liveness If on all links no application message is in transit, the DPE is triggered eventually at some process.

Local Liveness If on the links from and to a process no message is in transit, the DPE is triggered eventually at this process.

Permanent Liveness The DPE is triggered infinitely often at every process, and the times between two events is finite.

Note that by these definitions the DPE may also be triggered more often than necessary. Obviously, permanent liveness guarantees local liveness and local liveness guarantees global liveness. We denote the deadlock detectors defined by each of the properties with $\mathcal{DD}_G$, $\mathcal{DD}_L$, and $\mathcal{DD}_P$. A module that fulfills local (resp. global) liveness in the sense that an event is triggered *only* if there is no message is in transit is a much stronger device, however, and not covered by the considerations in this section. Such a property leads to the problem of *termination detection* [MFVP05].

As for failure detectors (cf. Section 3.2) we say a deadlock detector $\mathcal{DD}'$ is *reducible* to $\mathcal{DD}$ if there is a self-stabilizing asynchronous message-driven transformation algorithm $T_{\mathcal{DD}\rightarrow\mathcal{DD}'}$, such that $T_{\mathcal{DD}\rightarrow\mathcal{DD}'}$ uses $\mathcal{DD}$ and simulates $\mathcal{DD}'$. We denote this fact by $\mathcal{DD}' \preceq \mathcal{DD}$. By definition we have $\mathcal{DD}_G \preceq \mathcal{DD}_L \preceq \mathcal{DD}_P$. We now show that in a non-partitional

```
1   on receive msg from communication module
2       deliver msg to application
3       send (DpeMsg) to all via communication module
4       trigger DPE
5
6   on receive msg from application
7       send msg via communication module
8
9   on receive DPE from D_G
10      send (DpeMsg) to all
11      trigger DPE
12
13  on receive (DpeMsg) from communication module
14      trigger DPE
```

Figure 7.1: The transformation from $\mathcal{DD}_G$ to $\mathcal{DD}_P$

system, a deadlock detector fulfilling global liveness can be transformed into a deadlock detector with permanent liveness, rendering the three properties equivalent regarding computational power. For simplicity, in this case we assume a fully connected network.

Theorem 7.1. *$\mathcal{DD}_P \preceq \mathcal{DD}_G$ in a system that does not partition.*

Proof. We use the transformation algorithm of Figure 7.1. For simplicity this algorithm assumes a fully connected network. Since the algorithm has no local states only messages can be corrupted. All these messages are delivered by time $t_{GST} + \tau^+$, the algorithm has stabilized due to that. Thus we have to show the properties under process and link crashes only.

We show that for every time t there is a time $t' > t$ at which a DPE is triggered at process p, with $|t' - t| < \infty$. We distinguish two cases:

1. After t, there are always application messages in transit. Then on every reception of such a message at a process q, a DPE is triggered at p and by line `3`, (DpeMsg) is sent to all other processes, and by line `13`–`14` a DPE is triggered at every other process too, including p. Since all message transmission delays are finite, $|t'-t| < \infty$.

2. There is a time after which no application message is in transit anymore. Then at some process p, by the properties of $\mathcal{DD}_G$ a DPE from $\mathcal{DD}_G$ is triggered. By lines `10` and `13`–`14` also all other processes trigger a DPE. Since all message transmission delays are finite, and because of the properties of $\mathcal{DD}_G$, $|t' - t| < \infty$.

□

```
1  on receive msg from communication module
2      deliver msg to application
3      trigger DPE
4
5  on receive msg from application
6      send msg via communication module
7
8  on receive DPE from DD_L
9      trigger DPE
```

Figure 7.2: The transformation from $\mathcal{DD}_L$ to $\mathcal{DD}_P$

To adapt the transformation algorithm to sparse networks that do not partition is straightforward. For partitionable systems a deadlock detector that fulfills only global liveness cannot be used to obtain a local or permanent deadlock detector. Thus we only have the reducibility of local and permanent liveness in the partitionable case:

Theorem 7.2. *$\mathcal{DD}_P \preceq \mathcal{DD}_L$ in partitionable systems.*

Proof. We use the transformation algorithm in Figure 7.2. Assume by contradiction, that there is a time t and a process p, after which no DPE is triggered at p. If there is a time $t' > t$, after which no message is in transit from or to p, by local liveness of the $\mathcal{DD}_L$, the DPE is triggered by $\mathcal{DD}_L$, and by line `8-9`, the DPE is triggered at p, which is a contradiction. Thus there is a time after t, at which a message is in transit from or to p. If the message is in transit to p, it is eventually received, and by lines `1-3`, the DPE is triggered. If the message is in transit from p, it must have been sent by p, which can be the case only if p either received a message or received a DPE. In both cases (lines `1-3`, 8-9) we get a contradiction to the assumption that no DPE is triggered after t. □

Note that for both algorithms we do not aim at an efficient solution, since the transformation is only of theoretical interest: In practice a permanent deadlock detector is not more difficult to implement than a global or local deadlock detector.

For timing analysis of our algorithms we assume that the permanent deadlock prevention event is triggered at process p by time $t_{GST} + \eta$, where η is not known to processes. Note that the actual value of η has no influence on the correctness of our algorithms. Practically one could implement this e.g. with timers or clocks although any local mechanisms which gives some (even inaccurate) notion of elapsed time can be employed.

7.3 Impossibility Result

In this section we show the impossibility of implementing message-driven failure detectors with bounded memory. The intuitive argument is as follows: Due to the bounded memory assumption, every algorithm has to reuse messages and to run in cycles. Since they have no notion of real-time, they are not able to distinguish messages from a previous cycle from new ones. Initiated by a sufficiently large number of messages from the unstable period, it is hence possible that processes perceive a compressed notion of time. Since the failure detector is message-driven, this will trigger the generation of new messages according to this compressed time, which may go on perpetually. Obviously, since the failure detectors run much faster than they should, incorrect suspicions will occur. Due to self-stabilization and the bounded memory assumption they are not able to recover from this and thus suspect all processes forever.

The formal argument assumes by way of contradiction, that such a correct failure detector algorithm exists and constructs a cyclic execution of this algorithm. Such an execution must exist because of the bounded memory assumption. By excessively delaying messages, the failure detector can be forced to suspect every process at least once. By showing indistinguishability from a timely execution with messages from the unstable period we get the required contradiction.

Before t_{GST}, messages sent by some alive process p can be arbitrarily delayed by the adversary. The requirement of a failure detector implementation as well as self-stabilization and bounded memory requires that this process will be suspected within bounded time—since the other processes do not know that this happens before t_{GST}. If, after the suspicion, all of p's messages are immediately received, and p's messages are timely from then on, it must be deleted from the list of suspected processes, again within bounded time. Then the adversary may delay messages from some other process q.

We show in the proof of Theorem 7.3 that for any failure detector algorithm there exists a set of messages which are in transit at t_{GST} and received by processes adversarial such that the resulting execution is cyclic—due to the bounded memory requirement—and indistinguishable from the behavior described in the previous paragraph although all message delays are timely. This is possible, since processes do not have a sense of time and hence cannot distinguish this "compressed" execution from the former one. Thus, every correct processes is suspected infinitely often and eventually weak accuracy is violated; from this our impossibility result follows.

Theorem 7.3. *There is no deterministic message-driven self-stabilizing implementation of the eventually strong failure detector in a system with $\varepsilon > 0$, unknown channel capacity, and bounded memory of the failure detector, even assuming a deadlock detector that fulfills permanent liveness.*

Proof. Assume by contradiction that such an algorithm $\mathcal{A}$ exists. We first construct the following timed (but not timely; cf. Section 2.5) execution σ_0 of $\mathcal{A}$. Note that the adversary

can control the receive times of messages and the times of the deadlock prevention events in such a run.

1. No process ever crashes. We start in a configuration $C_1^{(1)}$. Our run proceeds in lock-step, i.e., the algorithm receives and sends messages only at times $t_k = k\tau_0$, with $\tau^- \leq \tau_0 \leq \tau^+$. All timely messages are sent at times t_k and received at times t_{k+1}. Note that at all times t_k no messages are in transit—except those received at time t_k—such that the configuration following t_k is just determined by the local states. We call the time between t_k and t_{k+1} a round.

2. For every $p \in \{1, \ldots, n\}$: Starting from a configuration $C_{2p-1}^{(1)}$, the adversary fires the deadlock prevention event at p, and all messages from and to process p are delayed (at least for one round), until some process suspects p. Such a configuration is reached eventually, since such a behavior is indistinguishable from a situation where p has crashed, and thus by strong completeness, p must eventually be suspected by some process. After that, the adversary delivers all delayed messages and we end up in a configuration $C_{2p}^{(1)}$. Between $C_{2p-1}^{(1)}$ and $C_{2p}^{(1)}$, p is suspected at least once. After $C_{2p}^{(1)}$, timing becomes correct for one round, resulting in configuration $C_{2p+1}^{(1)} = C_{2(p+1)-1}^{(1)}$.
   Finally, every process was suspected at least once by another process, and we end up in a configuration $C_1^{(2)} \triangleq C_{2n+1}^{(1)}$.

3. Starting repeatedly again from (2), we get chains of configurations $C_1^{(1)} \ldots C_{2n}^{(1)} \ldots C_1^{(i)} \ldots C_{2n}^{(i)}$. We call the run from $C_i^{(j)}$ to $C_{i+1}^{(j)}$ a phase and the run from $C_1^{(i)}$ to $C_{2n+1}^{(i)}$ an epoch.

4. Since every configuration $C_1^{(j)}$ depends only on the (finite) local states and the messages sent in $C_{2n}^{(j-1)}$ (which depend on the finite local states of $C_{2n}^{(j-1)}$), there are only a finite number of configurations $C_1^{(j)}$. Thus, there are numbers u and v ($u < v$), such that $C_1^{(u)} = C_1^{(v)}$. Let $E = u - v$ be the number of epochs. Every epoch has obviously $2n$ phases, where the number of rounds in phase ϕ of epoch e is $R(e, \phi)$, with $0 \leq \phi < 2n$, $0 \leq e < E$. The execution from $C_1^{(u)}$ to $C_1^{(v)}$ we refer to as cycle.

5. Since $C_1^{(u)}$ and $C_1^{(v)}$ are identical, the execution from $C_1^{(u)}$ to $C_1^{(v)}$ is applicable to itself. σ_0 is the execution that comprises Z cycles of the execution from $C_1^{(u)}$ to $C_1^{(v)}$, where Z has to be determined yet. The times t_k which have not been assigned yet are assumed such that σ_0 starts at time $t_0 = t_{GST} = 0$. $\sigma_{0,\infty} = \sigma_0\sigma_0 \ldots$ is the infinite run composed by infinitely many iterations of σ_0. Note that in $\sigma_{0,\infty}$, for every process and every time t there is a time $t' > t$, where it is suspected by some other process.

To summarize, we have the following notation for our execution σ_0:

name	symbol	number
cycle	z	Z
epoch	e	E
phase	ϕ	$2n$
round	r	$R(e,\phi)$

We define k to be a unique round number in σ_0 by the following function

$$k(z,e,\phi,r) = z\sum_{e'=0}^{E-1}\sum_{\phi'=0}^{2n-1} R(e',\phi') + \sum_{e'=0}^{e}\sum_{\phi'=0}^{2n-1} R(e',\phi') + \sum_{\phi'=0}^{\phi-1} R(e,\phi') + r. \tag{7.1}$$

It can easily be seen that $k(z,e,\phi,r)$ maps every tuple (z,e,ϕ,r) one-to-one to a value k, thus we can define the inverse functions $z(k)$, $e(k)$, $\phi(k)$ and $r(k)$. The times the algorithms start round r in phase ϕ of epoch e and cycle z is given by

$$t(z,e,\phi,r) = t_{k(z,e,\phi,r)} = k(z,e,\phi,r)\tau_0. \tag{7.2}$$

The total number of rounds in σ_0 is thus

$$R_{total} = Z\sum_{e=0}^{E-1}\sum_{\phi=0}^{2n-1} R(e,\phi). \tag{7.3}$$

Now we introduce a compressed run σ_1 that is indistinguishable from σ_0 for all processes. To that end, we define a function that maps all times t_k to times t'_k, such that $t'_{R_{total}} - t'_0 = \tau^-$, whereas the temporal order of the messages are preserved. Since the processes perceive only the message pattern and have no other time information, processes in the compressed execution σ_1 behave in the same way as in σ_0. The time transformation function is given by:

$$t'_k \;\triangleq\; f(t(z,e,\phi,r),\mathcal{A}) = \left(2nEz + 2ne + \phi + \frac{r}{R(e,\phi)}\right)\gamma, \tag{7.4}$$

where $\gamma \triangleq \frac{\tau^-}{2nEZ}$ gives the compressed length of a phase. Additionally, we chose $Z \geq \frac{\tau^-}{2nE\varepsilon}$, which implies $\gamma \leq \varepsilon$. The temporal order of the message delivery times is preserved, if $f(t_k,\mathcal{A})$ is a monotonically increasing function in t_k:

Lemma 7.1. *Function $f(t_k,\mathcal{A}) = (2nEz+2ne+\phi+\frac{r}{R(e,\phi)})\gamma$ is monotonically increasing in t_k.*

Proof. We first note that t_k is monotonically increasing in k, thus is suffices to show that $f(t_k)$ increases monotonically with k, i.e., $f(t_{k(z,e,\phi,r)}) < f(t_{k(z,e,\phi,r)+1})$. We distinguish four cases (recall that $r < R(e,\phi)$, $\phi < 2n$, and $e < E$):

- $r < R(e,\phi) - 1$. Then $k+1 = k(z,e,\phi,r+1)$ and the lemma is true.
- $r = R(e,\phi) - 1$ and $\phi < 2n-1$. Then $k+1 = k(z,e,\phi+1,0)$ and, because of $r < R(e,\phi)$ the lemma is true again.
- $r = R(e,\phi)-1$, $\phi = 2n-1$, and $e < E-1$. Then $k+1 = k(z,e+1,0,0)$, because of $\phi + r/R(e,\phi) < 2n$ the lemma holds.
- $r = R(e,\phi)-1$, $\phi = 2n-1$, and $e = E-1$. Then $k+1 = k(z+1,0,0,0)$. Since $2n(E-1)\phi + r/R(e,\phi) < 2nE$, the lemma holds.

Thus $f(t_k)$ is monotonically increasing. □

We now show that $\sigma_{1,\infty} = \sigma_1\sigma_1\ldots$ is indistinguishable from an infinite and timely execution $\sigma_{2,\infty} = \sigma_{2,0}\sigma_{2,1}\ldots$, if the system stabilizes at time $t_{GST} = 0$ with sufficiently many messages in the links at t_{GST}. The argument is as follows. During a single execution σ_1 all messages sent in σ_1 are received in σ_1. The following lemma constructs identical executions $\sigma_{2,i}$ where messages from $\sigma_{2,i-1}$ are received in $\sigma_{2,i}$ while the new messages are received in $\sigma_{2,i+1}$. Locally, temporal order of message receptions is maintained; thus σ_1 and $\sigma_{2,i}$ are indistinguishable.

Lemma 7.2. *For all processes, σ_1 is indistinguishable from an execution $\sigma_{2,i}$ that runs from $i\tau^-$ to $(i+1)\tau^-$, for any $i \geq 0$. At the end of $\sigma_{2,i+2}$ no messages from $\sigma_{2,i}$ are in transit. For $i > 0$, every message that is received in $\sigma_{2,i}$ satisfies $\tau^- \leq t_r - t_s \leq \tau^+$.*

Proof. By induction on i. For $i = 0$, recall that σ_0 has $2nEZ$ phases, and the duration of each phase is γ; thus $t'_{R_{total}} = \tau^-$. Hence, for every time t'_k a process receives a message in σ_1, the adversary delivers an identical message from the instable period in $\sigma_{2,0}$ at t'_k. Since all messages sent in $\sigma_{2,0}$ are delivered after τ^- and thus not within $\sigma_{2,0}$, $\sigma_{2,0}$ is indistinguishable from σ_1.

For the induction step $i > 0$, we assume that the lemma holds for $i-1$. For every message that is sent at t'_k in σ_1 and received at t'_ℓ in σ_1, by the induction hypothesis there is a send event at time $t_s = (i-1)\tau^- + t'_k$. The adversary delivers this message at time $t_r = i\tau^- + t'_\ell$. To see that this message is indeed timely, recall that—by the construction of σ_0—every message is delivered in the same phase in which it was sent, and by our compression function, the length of each phase in σ_1 is of length $\gamma \leq \varepsilon$. Thus, $t'_\ell - t'_k \leq \varepsilon$, and therefore

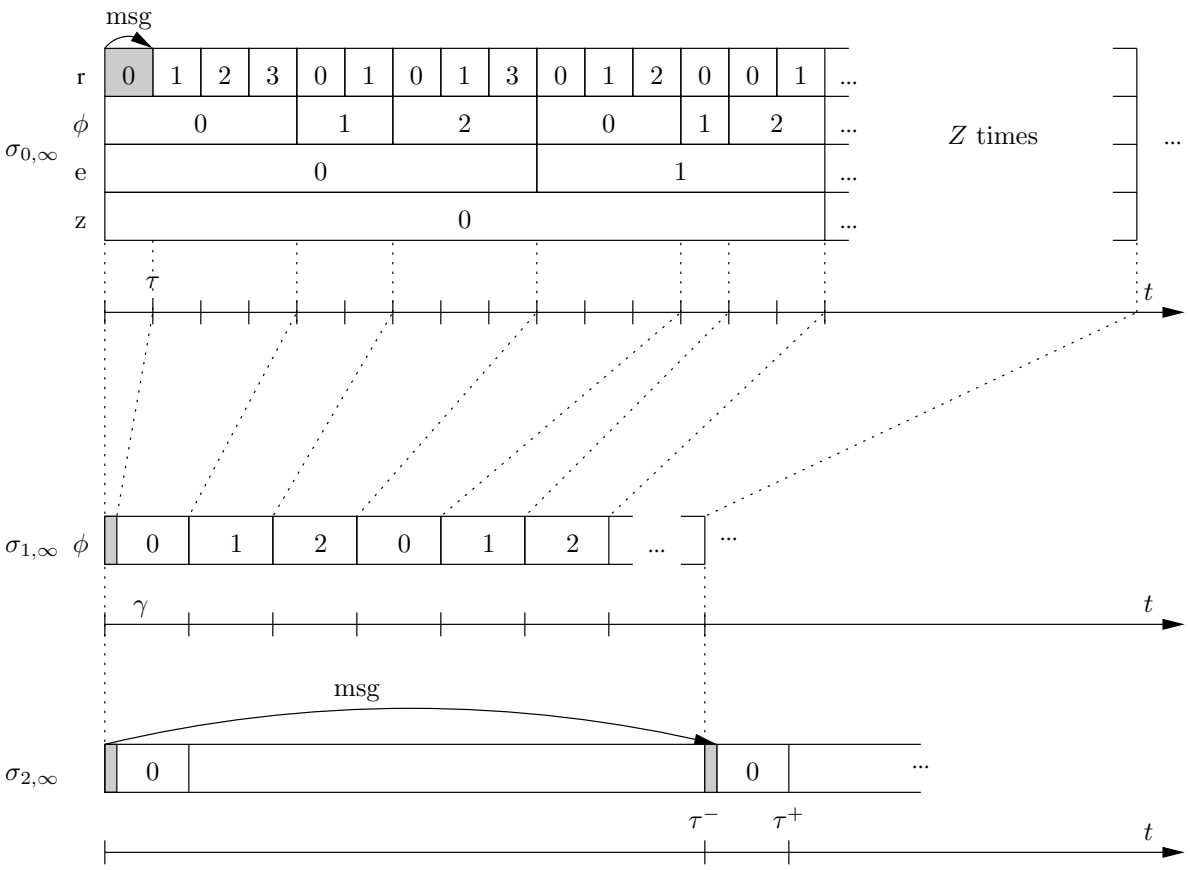

Figure 7.3: The construction of the executions $\sigma_{0,\infty}$, $\sigma_{1,\infty}$ and $\sigma_{2,\infty}$.

$t_r - t_s = (i\tau^- + t'_\ell) - ((i-1)\tau^- + t'_k) \leq \tau^+$. Since both, t'_k and t'_ℓ are nonnegative, $t_r - t_s = (i\tau^- + t'_\ell) - ((i-1)\tau^- + t'_k) \geq \tau^-$ follows trivially.

Since there are obviously no messages in transit from $\sigma_{2,i-2}$, and all messages from $\sigma_{2,i-1}$ are delivered in $\sigma_{2,i}$ on times $i\tau^- + t'_k$, the algorithm indeed behaves as in σ_1. □

Note that in $\sigma_{2,0}$ only messages from the unstable period are received, and all other messages fulfill $\tau^- \leq t_r - t_s \leq \tau^+$. Thus, the execution $\sigma_{2,\infty}$ is timely and indistinguishable from $\sigma_{1,\infty}$, which is indistinguishable from $\sigma_{0,\infty}$. In $\sigma_{0,\infty}$, however, no correct process ever stops from being suspected. The maximum number of messages (per process) from the unstable period is $M = n \cdot R_{total}$ and thus only depends on the algorithm, i.e., for any algorithm there is a channel capacity where there is a run where no correct process stops being suspected forever, although all processes are correct, which contradicts eventual weak accuracy. □

In the remainder of this chapter we show ways to circumvent our impossibility result.

```
1  state variables
2      ∀q ∈ Π :  lastmsg_p[q] ∈ ℕ
3
4  if received (p, k) from q
5      if k > lastmsg_p[q]
6          lastmsg_p[q] ← k
7          if k = max_{r∈Π}{lastmsg_p[r]} and ∄s, s ≠ q : lastmsg_p[s] = lastmsg_p[q]
8              suspect {r ∈ Π | k − lastmsg_p[r] ≥ Ξ}
9              send (p, k + 1) to all neighbors
10
11 if received (q, k) from q
12     send (q, k) to q
13
14 on deadlock-prevention-event do
15     send (p, max_{q∈Π}{lastmsg_p[q]} + 1) to all neighbors
```

Figure 7.4: Algorithm for process p with no bounds on number of messages on a link.

7.4 Unbounded Link Capacity

In this section we describe a novel implementation of $\diamond\mathcal{P}_\ell$ which handles an unbounded number of messages on the links, i.e., works also in systems where there is no known bound on $|Q(p, t_{GST})|$ for all processes p. This algorithm, however, requires unbounded memory.

The algorithm for a process p is given in Figure 7.4. With every neighbor of p, (p, k) messages are exchanged, where k is an integer. When a neighbor q receives such a message, q just returns it to p (lines `11-12`) and no further processing is done. For every neighbor q, p holds a variable $lastmsg_p[q]$, where it stores the highest integer k received in a (p, k) reply from q. We also use $lastmsg_{p,q}$ for $lastmsg_p[q]$. The highest value among all $lastmsg_{p,q}$ determines the *round* for process p. Thus we define

$$round_p(t) \triangleq \max_{q \in \Pi}\{lastmsg_{p,q}(t)\}$$

and

$$round'_p(t) \triangleq \max_{q \in \Pi}\{lastmsg'_{p,q}(t)\}$$

respectively.[2] Every time a new round is reached (by receiving a message (p, k) such that $k > round_p$), p sends a $(p, round_p + 1)$ message to all neighbors.

Note that the "fastest neighbor" determines the round progress, i.e., $round_p + 1$ is started when the first neighbor returns the $(p, round_p)$ message to p. By our timing

[2]Remember from Section 2.4 that $v'(t)$ denotes the value of a variable v after the step at time t.

model, this requires at least $2\tau^-$ time. The reply of the slowest neighbor requires at most $2\tau^+$. At this time, $round_p$ has reached at most $2\tau^+/2\tau^- < \Xi$ additional rounds. Thus, for every correct neighbor the difference $round_p - lastmsg_{p,q}$ is less than Ξ, whereas for every faulty neighbor p eventually stops updating $lastmsg_{p,q}$. So, we set $H(p,t)$ to the set of processes with $round_p - lastmsg_{p,q} \geq \Xi$ (line 8).

From time to time the last message is resent to every neighbor by line 14 in order to prevent a deadlock when messages are lost during the unstable period. Note that this has no influence on the operation of the algorithm, since all messages with $k \leq lastmsg_{p,q}$ are dropped, therefore only the first message that is received has an influence on the behavior of p. We assume that line 14 is activated at least once every η time steps. Note that this assumption is not required by the algorithm but just for the timing analysis. We start our analysis with some preliminary lemmata.

Lemma 7.3 (Monotonicity). *After t_{GST}, $round_p(t)$ is monotonically increasing with time t, i.e., $t_{GST} \leq t_1 \leq t \Rightarrow round_p(t_1) \leq round_p(t) \leq round'_p(t)$.*

Proof. Obvious, since $round_p$ is the maximum of all $lastmsg_{p,q}$, and by lines 5 and 6 $lastmsg_{p,q}$ is monotonically increasing. □

Lemma 7.4 (Progress). *There is a time t, $t_{GST} \leq t < t_{GST} + \max\{2\tau^+, \eta\}$, such that p broadcasts $(p, round'_p(t) + 1)$ at time t.*

Proof. At time t_{GST} we have to distinguish two cases:

1. There is at least one neighbor q of p, such that at least one message (p, ℓ), with $\ell > round_p(t_{GST})$ is in $Q(p,q,t)$. Let (p,k) be the first of them to be received by p at some time $t \geq t_{GST}$. Obviously, $t < t_{GST} + 2\tau^+$. Since by assumption this is the first message after t_{GST} that changes $round_p$, we have $round_p(t) = round_p(t_{GST})$. Thus, $lastmsg_{p,q}(t) \leq round_p(t) < k$, p executes lines 6 and 9 and hence broadcasts $(p, round'_p(t) + 1)$, with $round'_p(t) = k$.

2. No such message exists. Then by time $t = t_{GST} + \eta$, line 15 is executed, and also $(p, round_p(t) + 1)$ is broadcast.

Thus by time $t_{GST} + \max(2\tau^+, \eta)$ the required message is broadcast in both cases. □

Lemma 7.5 (Stabilization). *For every message (p,k), which is received by p at some time $t \geq t_{GST} + 2\tau^+$, it holds that $k \leq round_p(t) + 1$.*

Proof. Since sending a message to a neighbor and receiving the answer takes at most $2\tau^+$, there is a time $t_1 \geq t_{GST}$ when p has broadcast (p,k). However, since we are after t_{GST}, k must be equal to $round'_p(t_1) + 1$, since p can broadcast only $(p, round'_p(t_1) + 1)$ messages (lines 9,15). By Lemma 7.3, $round_p$ is monotonically increasing with time. Therefore from $t \geq t_1$ follows $round'_p(t) + 1 \geq round_p(t) + 1 \geq round_p(t_1) + 1 = k$. □

Let $t_{stable} \triangleq t_{GST} + \max(2\tau^+, \eta)$ denote the time by which we have progress (Lemma 7.4) and correct message pattern (Lemma 7.5).

Lemma 7.6 (Fastest Progress). *Let correct process p broadcast (p, k) at some time $t \geq t_{stable}$ for the first time. Then p does not broadcast $(p, k + \ell)$ before time $t + 2\ell\tau^-$.*

Proof. By induction on ℓ. For $\ell = 1$ assume by contradiction that p broadcasts $(p, k + 1)$ before time $t + 2\tau^-$. If $(p, k + 1)$ is sent by line `15` it was sent also by line `9` before (since after t_{stable} by Lemma 7.4 at least one message was sent by line `9`), which would not be the first time. Since $(p, k+1)$ is sent by line `9`, p received a (p, k) message which was sent before time $t+\tau^-$ by some q (as response – line `11`) and before time t by p. Contradiction.

Now assume p broadcasts $(p, k + \ell - 1)$ not before $t + 2(\ell - 1)\tau^-$ the first time. By the same argument, p does not broadcast $(p, k + \ell)$ before time $t + 2\ell\tau^-$. □

Lemma 7.7 (Slowest Progress). *Let correct process p broadcast (p, k) at some time $t > t_{stable}$ for the first time. p broadcasts $(p, k + \ell)$ by $t + 2\ell\tau^+$.*

Proof. By induction on ℓ. Since p broadcasts (p, k) at time t, all neighbors of p receive this message by time $t+\tau^+$ and every correct neighbor (by our system model, there exists at least one) returns it. These messages are received by p by time $t + 2\tau^+$. Consider the time of the reception of the first of these messages. Because of Lemma 7.5 (note that $k = round'_p(t)+1$), p receives no message (p, k') with $k' > k$ by $t+2\tau^+$, thus p broadcasts $(p, k + 1)$.

For $\ell > 1$, assume p broadcasts (p, k) by time $t + 2(\ell - 1)\tau^+$. By the same argument, p does broadcasts $(p, k + \ell)$ by time $t + 2\ell\tau^+$. □

Lemma 7.8. *For every time $t > t_{stable} + 2\tau^+\Xi$ and every correct neighbor q of correct process p, it holds that $round'_p(t) - lastmsg'_{p,q}(t) < \Xi$.*

Proof. Since all variables are non-decreasing, the condition could be only violated by increasing $round_p$. So we consider only times t, where $round'_p(t) > round_p(t)$. By Lemma 7.5 we have $round'_p(t) = round_p(t) + 1$. By Lemmata 7.4, 7.5 and 7.6, a message $(p, round'_p(t)-\Xi+1)$ was broadcast by p at time $t_s \leq t-2\tau^-\Xi$, by Lemma 7.7, $t_s > t_{stable}$, so this message really exists. The reply to this message is received from every correct neighbor q by time $t_r \leq t_s+2\tau^+$, thus $lastmsg'_{p,q}(t_r) \geq round'_p(t)-\Xi+1 > round'_p(t)-\Xi$. Because of $\Xi > \Theta$ we have $t_r \leq t_s + 2\tau^+ \leq t - 2\tau^-\Xi + 2\tau^+ \leq t$. Since by the algorithm $lastmsg'_{p,q}$ is monotonically increasing it follows that $lastmsg'_{p,q}(t) \geq lastmsg'_{p,q}(t_r)$. Hence we get $round'_p(t) - lastmsg'_{p,q}(t) < \Xi$. □

Theorem 7.4 (Local Completeness). *Eventually every non-correct neighbor of p is suspected by p.*

Proof. Let t_{crash} be the time q crashes, and $t = \max\{t_{crash}, t_{GST}\}$. Then no message from q to p is received after $t + \tau^+$. After this, $lastmsg_{p,q} \leq round_p$ remains unchanged. By Lemma 7.7, $round_p$ reaches $round_p(t) + \Xi$ by time $\max\{t_{crash} + \tau^+, t_{stable}\} + 2\Xi\tau^+$. Since $round_p$ is nondecreasing, q remains suspected from then on. □

Theorem 7.5 (Eventual Local Accuracy). *Eventually p stops suspecting every correct neighbor of p.*

Proof. Follows directly from Lemma 7.8 and line 8 of the algorithm. □

Corollary 7.1. *The algorithm in Figure 7.4 implements a* self-stabilizing eventually perfect local failure detector *in a sparse network.*

After stabilization, a crashed process is suspected Ξ rounds after it crashed, i.e., the worst case failure detection time is $2\Xi\tau^+$.

7.5 Bounded Link Capacity

In this section we give a solution to failure detection which requires that the number of messages which can be in transit at the same time over one link is bounded, and this bound is known in advance. In contrast to the algorithm of the previous section, this one requires just bounded memory size. We believe that this result is of practical interest. Real computers have bounded memory, which is not only used to store variables of our algorithms, but also to store messages in various queues. Since, queues are the significant parts of links, the assumptions that the number of messages are bounded seems reasonable to us.

The algorithm is depicted in Figure 7.5. In concept, it works similar to the one in the last section. However, since our integers are bounded, eventually we need to wrap-around the round number. We call one such cycle a *phase*. To avoid that messages of the previous phase interfere with the current one, we use phase numbers. Since also the range of the phase numbers is bounded due to the bounded memory assumption, we have to ensure that there are sufficiently many distinct phase numbers such that no interference is possible. We show in our analysis, that if there are at most M messages in all links of a process, $M + 2$ phases are sufficient to ensure stabilization. The idea behind this is that there exists at least one phase which cannot be shortened by faulty messages from the unstable period. The second difference to the algorithm in Figure 7.4 is that this algorithm broadcasts only on a phase switch, whereas the previous one broadcasts every round.

By our assumption, $|Q(p, t_{GST})| \leq M < \infty$ for all processes p. For any phase number ph we further define $next(ph) \triangleq (ph + 1) \mod (M + 2)$ and $prev(ph) \triangleq (ph + M + 1) \mod (M + 2)$.

Lemma 7.9. *For every process p, in any execution of our algorithm, there exists at least one phase number ph_0, such that no message (p, ph_0, k) is in $Q(p, t_{GST})$ and $ph_0 \neq phase_p(t_{GST})$.*

```
1  state variables
2      phase_p ∈ {0, ..., M + 1}
3      ∀q ∈ Π : lastmsg_p[q] ∈ {0, ..., Ξ}
4
5  if received (p, ph, k) from q
6      if ph = phase_p and k > lastmsg_p[q]
7          if k < Ξ
8              lastmsg_p[q] ← k
9              send (p, phase_p, k + 1) to q
10         else
11             suspect {r ∈ Π | lastmsg_p[r] = 0}
12             phase_p ← (phase_p + 1) mod (M + 2)
13             ∀r ∈ Π : lastmsg_p[r] ← 0
14             send (p, phase_p, 1) to all neighbors
15
16 if received (q, ph, k) from q
17         send (q, ph, k) to q
18
19 on deadlock-prevention-event do
20     ∀q ∈ Π : send (p, phase_p, lastmsg_p[q] + 1) to q
```

Figure 7.5: Algorithm for process p with known upper bound on number of messages in transit on links.

Proof. Obviously, $|Q(p, t_{GST})| = x \leq M$. At time t_{GST} process p can be in one phase only. The number of phase numbers is $M + 2 > x + 1$ such that at least 1 phase number remains. □

We now give two properties, that define the legitimate states for process p. In more detail, when the stability property holds, it is ensured that there are no "malicious" messages from before t_{GST} in transit. The progress property ensures that the system is not deadlocked, i.e., that there are sufficiently many messages in transit to keep the failure detector working.

Definition 7.1 (Stability). *For a process p, the predicate $\mathcal{PS}(p,t)$ holds at time t iff there is no $next(phase_p(t))$ message in transit. Formally,*

$$\mathcal{PS}(p,t) \equiv \nexists k : \ (p, next(phase'_p(t)), k) \in Q(p,t)$$

Definition 7.2 (Progress). *For a process p, the predicate $\mathcal{PP}(p,t)$ holds at time t iff there is at least one correct neighbor q of p, from or to which a message with $k > lastmsg_{p,q}(t)$ and current phase is in transit. Formally,*

$$\mathcal{PP}(p,t) \equiv \exists q \in (\mathcal{C} \cap \mathrm{nb}(p)) \ \exists k > lastmsg_{p,q}(t) : \ (p, phase_p(t), k) \in Q(p,q,t)$$

Intuitively, if $\mathcal{PS}(p,t)$ holds, there are no more erroneous messages in transit. $\mathcal{PP}(p,t)$ indicates whether there are any relevant messages in transit and thus we have progress.

We start by showing closure of progress, i.e., if $\mathcal{PP}$ holds once after t_{GST} it holds forever.

Lemma 7.10. *If there is a time $t_0 \geq t_{GST}$, where $\mathcal{PP}(p,t_0)$ holds, then $\mathcal{PP}(p,t)$ holds also for all times $t > t_0$. Formally,*

$$\exists t_0 \geq t_{GST} : \ \mathcal{PP}(p,t_0) \Rightarrow \forall t > t_0 \ \mathcal{PP}(p,t)$$

Proof. Assume by contradiction that there is a time $t > t_0$, where $\mathcal{PP}(p,t)$ does not hold anymore for the first time. Since by assumption the predicate held before that, for some non-faulty q, either $lastmsg_{p,q}(t) \neq lastmsg'_{p,q}(t)$ or $(p, phase_p(t), k) \notin Q'(p,q,t)$ anymore. In both cases, p has received a $(p, phase_p(t), k')$ message with $k' > lastmsg_{p,q}(t)$ and thus sends either a $(p, phase'_p(t), lastmsg'_{p,q}(t)+1)$ or sends a $(p, phase'_p(t), round'_p(t)+1)$ message and sets $lastmsg_{p,q} = 0$. In both cases the property holds also at time t. Contradiction. □

Lemma 7.11. *By time $t_{GST} + \eta$, $\mathcal{PP}(p,t)$ holds forever.*

Proof. By time $t_{GST}+\eta$ the deadlock prevention event is triggered and p sends a $(p, phase_p, lastmsg_{p,q}+1)$ to every neighbor q. By our system model, at least one of them is non-faulty and thus $\mathcal{PP}$ holds for p at this time. By Lemma 7.10 after that $\mathcal{PP}$ holds forever. □

We have seen that our algorithm stabilizes such that $\mathcal{PP}$ always holds after bounded time after t_{GST}. We now turn our attention to the $\mathcal{PS}$ property and start with some preliminary lemmata.

Lemma 7.12 (Fastest Progress). *Assume p starts phase $ph = phase_p(t)$ by broadcasting $(p, ph, 1)$ at time t, and $\mathcal{PS}(p,t)$ holds. Then p does not broadcast $(p, next(ph), 1)$ before time $t + 2\tau^- \Xi > t + 2\tau^+$*

Proof. p broadcasts $(p, next(ph), 1)$ only if it receives a (p, ph, Ξ) message from one of its neighbors. We show by induction on Ξ that this is the case not before $t + 2\tau^-\Xi$. For $\Xi = 1$, sending (p, ph, Ξ) to some neighbor q and receiving the answer takes at least time $2\tau^-$, and since by $\mathcal{PS}(p,t)$ no other messages with phase ph are in transit at time t, p cannot receive $(p, ph, 1)$ before $t + 2\tau^-$. Because p is still in phase ph, no message with other phases were broadcast, thus $\mathcal{PS}$ still holds. Assume p receives a $\Xi - 1$ message not before $t + 2\tau^-(\Xi - 1)$. By the same argumentation, this message is not received before $t + 2\tau^-\Xi$ and $\mathcal{PS}$ holds. Since $\Xi > \Theta$ (compare Section 2.5), $t + 2\tau^-\Xi > t + 2\tau^+$. □

Lemma 7.13 (Slowest Progress). *Assume p starts phase ph by broadcasting $(p, ph, 1)$ at time t. Then p broadcasts $(p, next(ph), 1)$ by $t + 2\tau^+\Xi$.*

Proof. Note that p broadcasts $(p, next(ph), 1)$ if it receives a (p, ph, Ξ) message from one of its neighbors and is still in phase ph. If p is no more in phase ph we are done, so we show by induction on Ξ that p receives a (p, ph, Ξ) message by time $t + 2\tau^+$. Sending a message to a neighbor and back requires at most time $2\tau^+$, thus by time $t + 2\tau^+$ receives $(p, ph, 1)$. For the inductive step assume p receives $(p, ph, \Xi - 1)$ by time $t + 2\tau^+(\Xi - 1)$. Then by the same argument p receives (p, ph, Ξ) by time $t + 2\tau^+\Xi$. □

Lemma 7.14. *Assume $phase_p(t) = ph$. Then $phase_p(t_1) = prev(ph)$ for some times $t_1 > t > t_{GST}$ only if p was in all other phases in the time interval $[t, t_1]$.*

Proof. By line 12 of the algorithm, p changes its phase only to $next(phase_p(t))$ and thus has to adopt all other values before reaching $prev(phase_p(t))$. □

Lemma 7.15. *$\mathcal{PS}(p, t)$ holds at time $t_{GST} + 2\tau^+$.*

Proof. We have to show that no messages (p, ℓ, k) for some k and $\ell = next(phase_p(t))$ is in transit at time $t = t_{GST} + 2\tau^+$. Obviously, no message which is in transit at time t was already in transit at time t_{GST}. Moreover, no message which is in transit at time t is a reply from one of p's neighbors to a possibly faulty message which was in $Q(p, t_{GST})$ since all these responses must have been received by p before t. Thus, message (p, ℓ, k) can only be in transit at time t if p was in phase ℓ at some time t_1, $t_{GST} \leq t_1 \leq t$. It remains to show that this is not possible.

As p is in phase $prev(\ell)$ at time t it must, by Lemma 7.14, have been in all phases $(0..M + 1)$ between t_1 and t, thus there must be some time t_2, $t_1 \leq t_2 \leq t$ such that $phase_p(t_2) = prev(ph_0)$, i.e., phase ph_0 from Lemma 7.9 was started then. Thus $\mathcal{PS}(p, t_2)$. By Lemma 7.12 this phase cannot be terminated before some time $t_3 > t_2 + 2\tau^+ \geq t$ which is a contradiction to p being in phase $prev(\ell)$ at time t. □

It remains to show closure over $\mathcal{PS}$, i.e., if $\mathcal{PS}$ is reached once, it holds forever.

Definition 7.3. *We define $\sigma(p, t, ph)$ as the first time after t, where p reaches phase ph. Formally,*

$$\sigma(p, t, ph) \triangleq \min\{t' > t \mid phase_p(t') = ph\}$$

Lemma 7.16. *From $\mathcal{PS}(p, t)$ where $t \geq t_{GST}$ follows that $\mathcal{PS}(p, t')$ holds for all times t', $t \leq t' < \sigma(p, t, next(phase_p(t)))$.*

Proof. Since $phase_p$ remains unchanged, no spontaneous messages are generated after t_{GST} and p sends $phase_p(t)$ messages only. □

Lemma 7.17. *Let $\mathcal{PS}(p,t)$ hold at time $\sigma(p,t,ph)$ where $t \geq t_{GST}$. Then $\mathcal{PS}(p,t')$ at time $t' = \sigma(p,t,next(ph))$.*

Proof. By Lemma 7.12 p terminates phase ph after $\sigma(p,t,ph) + 2\tau^+$. All messages which are in transit to p at time $\sigma(p,t,ph)$ are received by time $\sigma(p,t,ph)+\tau^+$. All messages for other phases than ph are ignored by p (and hence no messages are sent). All messages for phases other than ph which are in transit from p to its neighbors are answered by them by line 17. The answers are received by p by $\sigma(p,t,ph) + 2\tau^+$ and ignored as well since p is still in phase ph. Thus, no messages for other phases than ph are in transit at time t'. Since $next(next(ph)) \neq ph$ the lemma holds. □

Lemma 7.18. *After time $t_{stable} = t_{GST} + 2\tau^+$, $\mathcal{PS}$ holds at all phase switch times.*

Proof. By Lemma 7.15 $\mathcal{PS}(p,t)$ holds at time $t = t_{GST} + 2\tau^+$. By Lemma 7.16 it follows that $\mathcal{PS}(p,t')$ holds for all times t', $t \leq t' < \sigma(p,t,next(phase_p(t)))$. From an inductive application of Lemma 7.17 it follows that $\mathcal{PS}$ holds at all phase switch times after that. □

From these lemmata it follows that after some time, all phases are sufficiently long to timeout processes. Thus we show local completeness and local accuracy in the following.

Theorem 7.6 (Local Completeness). *Eventually every non-correct neighbor of p is suspected by p.*

Proof. Assume neighbor q of p has crashed. By Lemma 7.11, $\mathcal{PP}$ holds by time $t_{GST} + \eta$. Note that every message (p, ph, k) from a correct neighbor r, with $k > lastmsg_{p,r}$ and $ph = phase_p$ causes either a message $(p, ph, k+1)$ (for $k < \Xi$) or a $(p, ph+1, 1)$ message. Consequently, eventually p reaches $k = \Xi$ and switches to the next phase (lines 11-14). When p reaches $k = \Xi$ in the next phase, $lastmsg_{p,q} = 0$, since there was no message from q. According to line 11, p suspects q. □

Theorem 7.7 (Eventual Local Accuracy). *Eventually p stops suspecting every correct neighbor of p.*

Proof. By Lemma 7.18 and Lemma 7.12 all phases that are started after $t_{GST} + 2\tau^+$ are longer than $2\tau^+$. This is sufficiently long for all answers of correct process p's correct neighbors q to p's $(p, ph, 1)$ message are received by p before it executes line 11 at some time t. It follows $lastmsg_{pq}(t) > 0$ for every correct neighbor q when p executes line 11 such that no correct processes will ever be suspected by p. □

Corollary 7.2. *The algorithm in Figure 7.5 implements a* self-stabilizing eventually perfect local failure detector *in a sparse network.*

When a process crashes in a phase (after replying at least one message) it is suspected at the end of the next phase. A full phase takes at most $2\Xi\tau^+$ time, and a the time from the first reply of a message to the end of a phase is $(2\Xi - 1)\tau^+$. Thus the worst case failure detection time—once the failure detector has stabilized—is $(4\Xi - 1)\tau^+$.

7.6 Randomization

The impossibility result of Theorem 7.3 considers *deterministic* algorithms. In this section we present a randomized solution for our problem. Note that this algorithm differs from the deterministic bounded memory algorithm of Section 7.5 only in line `12` and in the number of phases.

For both algorithms, the critical point during stabilization is that "malicious" messages from the instable period may shorten the duration of phases and so the algorithm looses its time base. In contrast to the deterministic bounded memory algorithm of Section 7.5 which requires one additional phase per malicious message, the randomized algorithm achieves stabilization by choosing a random phase number every time a new phase is started. Since messages with a wrong phase number are dropped, there is a nonzero chance for each malicious message to be dropped. Thus the probability of stabilization converges to 1. Note that once stabilization is reached, it is preserved, since then every phase takes at least $2\tau^+$ and every message with a phase number distinct to the correct one is dropped in such a phase. In this section, the number M of messages in transit at t_{GST} is used for analysis only and neither bounded nor known to the algorithm.

As we just want to explore the boundaries of the impossibility result, we do not aim at an optimal solution regarding probability of stabilization here.

For a fixed execution of our algorithm let us denote with $next(phase_p(t))$ the value of $phase_p$ after the next coin toss following t. Like in the last section we define the following properties:

Definition 7.4 (Stability). *For a process p, the predicate $\mathcal{PS}(p,t)$ holds at time t iff there is no $next(phase_p(t))$ message in transit. Formally,*

$$\mathcal{PS}(p,t) \equiv \nexists k: \ (p, next(phase'_p(t)), k) \in Q(p,t)$$

Definition 7.5 (Progress). *For a process p, the predicate $\mathcal{PP}(p,t)$ holds at time t iff there is at least one correct neighbor q of p, from or to which a message with $k > lastmsg_{p,q}(t)$ and current phase is in transit. Formally,*

$$\mathcal{PP}(p,t) \equiv \exists q \in (\mathcal{C} \cap \mathrm{nb}(p)) \ \exists k > lastmsg_{p,q}(t): \ (p, phase_p(t), k) \in Q(p,q,t)$$

Since the changes in the algorithm have no impact on progress, we can adopt from Section 7.5:

Lemma 7.19. *Eventually $\mathcal{PP}(p,t)$ holds forever.*

It remains to show that (with high probability) the algorithm stabilizes:

Lemma 7.20. *By an expected time of at most $(\Xi+1)\tau^+ 2^{M+1}$ after t_{GST}, $\mathcal{PS}$ holds forever.*

```
1  state variables
2      phase_p ∈ {0, 1, 2}
3      ∀q ∈ Π : lastmsg_p[q] ∈ {0, ..., Ξ}
4
5  if received (p, ph, k) from q
6      if ph = phase_p and k > lastmsg_p[q]
7          if k < Ξ
8              lastmsg_p[q] ← k
9              send (p, phase_p, k + 1) to q
10         else
11             suspect {r ∈ Π | lastmsg_p[r] = 0}
12             phase_p ← coin({0, 1, 2} − {phase_p})
13             ∀r ∈ Π : lastmsg_p[r] ← 0
14             send (p, phase_p, 1) to all neighbors
15
16 if received (q, ph, k) from q
17     send (q, ph, k) to q
18
19 on deadlock-prevention-event do
20     ∀q ∈ Π : send (p, phase_p, lastmsg_p[q] + 1) to q
```

Figure 7.6: Randomized failure detector Implementation with Bounded Memory

Proof. We use the proof technique of a *game between the adversary and luck* of [Dol00]. Both players have full knowledge of the system and of the execution history. Every time a coin toss is performed, luck can intervene and determine the result. On the other hand, the adversary has all other choices. The bound expected time of stabilization is r/cp, where r is the worst case time to win the game, and cp the combined probability of all coin toss results, where luck did intervene.

Luck's strategy is as follows: W.l.o.g. at t_{GST} we are in phase 0. At the first coin toss, luck does not intervene. W.l.o.g. we are now in phase 1. For every following coin toss (until the algorithm stabilizes), luck chooses either phase 1 or 2, depending on which one is possible. So by time $2\tau^+$ no phase-0 messages are in transit anymore. After this, the next time luck intervenes (which is after at most $2\Xi\tau^+$ time steps) it chooses phase 0 and the algorithm stabilizes. Note that M malicious messages may cause at most M coin tosses before $2\tau^+$. The combined probability of all interventions of luck is $cp = (1/2)^{M-1} \cdot 1/2 = (1/2)^M$ and the time to win the game is $r = (2\Xi + 2)\tau^+$. Thus the expected time for stabilization is $r/cp = (\Xi + 1)\tau^+ 2^{M+1}$. □

The following lemmata consider the stabilized algorithm, their proofs are thus the same as in Section 7.5.

Lemma 7.21 (Fastest Progress). *Assume p starts phase $ph = phase_p(t)$ by broadcasting $(p, ph, 1)$ at time t, and $\mathcal{PS}(p,t)$ holds. Then p does not broadcast $(p, next(ph), 1)$ by time $t + 2\tau^-\Xi > t + 2\tau^+$*

Lemma 7.22 (Slowest Progress). *Assume p starts phase ph by broadcasting $(p, ph, 1)$ at time t. Then p broadcasts $(p, next(ph), 1)$ by $t + 2\tau^+\Xi$.*

With these lemmata, we can show the properties of our failure detector:

Theorem 7.8 (Local Completeness). *Eventually every non-correct neighbor of p is suspected by p.*

Proof. Assume neighbor q of p has crashed. By Lemma 7.19, $\mathcal{PP}$ eventually holds. Note that every message (p, ph, k) from q, with $k > lastmsg_{p,q}$ and $ph = phase_p$ causes either a message $(p, ph, k+1)$ (for $k < \Xi$) or a $(p, ph', 1)$ message with $ph' \neq ph$. Consequently, eventually p reaches $k = \Xi$ and switches to a new phase (lines `11-14`). When p reaches $k = \Xi$ in the new phase, $lastmsg_{p,q} = 0$, since there was no message from q. According to line `11`, p suspects q. □

Theorem 7.9 (Eventual Local Accuracy). *Eventually p stops suspecting every correct neighbor of p.*

Proof. By Lemma 7.20 and Lemma 7.21 all phases that are started after $t_{GST} + 2\tau^+$ are longer than $2\tau^+$. This is sufficiently long for all answers of correct process p's correct neighbors q to p's $(p, ph, 1)$ message to be received by p before it executes line `11` at some time t. It follows that $lastmsg_{pq}(t) > 0$ for every correct neighbor q when p executes line `11` so that no correct processes will ever be suspected by p. □

Corollary 7.3. *The algorithm in Figure 7.6 implements a randomized time free self-stabilizing implementation of $\diamond\mathcal{P}_\ell$ for systems with unbounded link capacity and bounded memory. It has an expected stabilization time of at most $(\Xi + 1)\tau^+ 2^{M+1}$, where M is the (unbounded) number of messages from the instable period.*

7.7 No Timing Uncertainty

Another condition of Theorem 7.3 is $\varepsilon > 0$. In this Section we show that this is a necessary condition for the impossibility by providing an algorithm for $\varepsilon = 0$, that is $\tau^+ = \tau^- \stackrel{\Delta}{=} \tau$.

Executing the algorithm given in Figure 7.7, process p just sends (p) messages to all its neighbors. And these neighbors just reply (p) upon reception. Because of the extremely strong timing assumption $\varepsilon = 0$, all replies must reach p simultaneously; upon reception of a set of messages at time t, p thus suspects all processes where no messages are received at time t.

```
1  if received (p) from some set of processes S
2      suspect Π − S
3      send (p) to all neighbors
4
5  if received (q) from q
6      send (q) to q
7
8  on deadlock-prevention-event
9      send (p) to all neighbors
```

Figure 7.7: Failure detector implementation for no timing uncertainty

Theorem 7.10. *The algorithm in Figure 7.7 is a message-driven self-stabilizing implementation of the $\diamond\mathcal{P}_\ell$ failure detector for systems where $\varepsilon = 0$, with unbounded link capacity and bounded memory.*

Proof. We show the properties of an eventually perfect failure detector:
Local Completeness: If a process q crashes at time t, every other correct process p does not receive any messages from q anymore, thus every time line `2` is executed after $t + \tau$, q is suspected (and thus remains suspected forever). However, since eventually p executes line `9`, and at least one correct neighbor replies by lines `5-6`, eventually line `2` is executed.
Eventual Local Accuracy: Eventually p executes line `9`, we denote this time by t. Thus, at time $t+\tau$, every other correct process q receives and answers the message by line `5-6`. At time $t + 2\tau$, p receives message (p) from process q and therefore does not suspect q. □

Note that—again—we do not aim at an optimal solution, since $\varepsilon = 0$ is unrealistic and thus the result is only of theoretical interest.

7.8 From Local to Global Failure Detection

In this section we show that an eventually local failure detector can be transformed into an eventually perfect failure detector (for partitionable systems) by an asynchronous message-driven self-stabilizing algorithm. This shows that also for message-driven algorithms, an eventually local failure detector is of the same power than an eventually perfect one. Again, this algorithm needs a deadlock detector to avoid blocking of the algorithm if all messages get lost.

This transformation differs from the one in Section 6.5 in that it is message-driven and does not preserve stability. The transformation is simple: the local suspect lists are exchanged between the processes, the processes build their global suspect list by intersecting all estimates. A hop counter is used to ensure that no old message circulate forever in the system.

```
1  input localsuspect_p ⊆ nb(p)
2  output globalsuspect_p ⊆ Π
3
4  state variables
5      ∀q ∈ Π :  suspect_p[q] ⊆ Π
6
7  on receive (q, S, h) from a neighbor
8      suspect[q] = S
9      globalsuspect_p = ⋂_{q∈Π} suspect[q]
10     if h > 0 and h < N − 1
11         send (q, s, h − 1) to all neighbors
12
13 on receive DPE from DD_P or if localsuspect_p changes
14     suspect_p[p] = localsuspect_p
15     send (p, suspect_p[p], n − 2) to all neighbors
```

Figure 7.8: The transformation from $\diamond\mathcal{P}_\ell$ to $\diamond\mathcal{P}$.

.

Theorem 7.11. *The algorithm in Figure 7.8 is a self-stabilizing transformation from $\diamond\mathcal{P}_\ell$ to $\diamond\mathcal{P}$ using bounded local space.*

Proof. We show the properties of an eventually local failure detector:
(Strong Completeness:) Suppose process p crashes. Then according to the properties of $\diamond\mathcal{P}_\ell$, there is a time t_1, where every correct neighbor of p permanently suspects p. After that, every message $(q, S, n-2)$ that a neighbor q of p issues, has $q \in S$. Since all transmission delays are finite, there is a time $t_2 \geq t_1$, where no message (q, S, h) with $q \in \text{nb}(p)$ and $p \notin S$ is in transit anymore. By the properties of the permanent deadlock detector module, at some time $t_3 > t_2$ DPE is triggered and $(p, S, N-2)$ is flooded over the network. Since no other message (q, S, h) with $q \in \text{nb}(p)$ and $p \notin S$ is in transit and no such message is generated anymore, eventually every correct process r has $p \in suspect_r[q]$ for all $q \in \text{nb}(p)$. For $q \notin \text{nb}(p)$, $suspect_r[q]$ trivially contains p, thus every process r eventually suspects p.

(Eventual Strong Accuracy:) Assume process p never crashes. Then according to the properties of $\diamond\mathcal{P}_\ell$, there is a time t_1, after which no neighbor of p suspects p. After that, no message $(q, S, N-2)$ that a neighbor q of p issues, has $q \in S$. Since all transmission delays are finite, there is a time $t_2 \geq t_1$, where no message (q, S, h) with $q \in \text{nb}(p)$ and $p \in S$ is in transit anymore. By the properties of the permanent deadlock detector module, at some time $t_3 > t_2$ DPE is triggered and $(p, S, n-2)$ is flooded over the network. Since no message (q, S, h) with $q \in \text{nb}(p)$ and $p \in S$ is in transit and no such message is generated anymore, eventually every correct process r has at least one $q \in \text{nb}(p)$ such

that $p \notin suspect_r[q]$. After that, every process r eventually forever stops suspecting p. □

Note that we do not aim for an efficient solution for this problem here.

7.9 Discussion

This chapter investigated message-driven self-stabilizing implementations of failure detectors. Intuitively, the major problem stems from the requirement of message-driven algorithms. In contrast to time-driven algorithms [BKM97, DLS88], where local clocks can be employed to periodically send messages independently of the rate of received messages, message-driven algorithms can only react to received messages. The time between send events thus depends just on the incoming message pattern. Due to arbitrary system states (self-stabilization requirement), perceived time can be compressed arbitrarily such that the message pattern provides unreliable time information. This leads to our impossibility result in Section 7.3 which shows that there is no message-driven deterministic self-stabilizing implementation of unreliable failure detectors in systems with timing uncertainty where the link capacity is unknown and the memory of the processes is bounded.

However, there are ways to circumvent the impossibility. Section 7.4 presents a simple solution, which requires unbounded memory—an assumption which is not reasonable when considering self-stabilizing algorithms for implementations in real systems. We therefore presented a practical solution in Section 7.5 which requires bounded memory. This solution can be used in real systems, as the crucial value M—a bound on the number of messages in the links—may be chosen in such a way that it is larger that the real number of messages in real systems (recall the space requirement is just logarithmic in M).

Finally we presented two other algorithms which show that it is possible to solve the problem by randomization and in system without timing uncertainty. It is thus possible to implement $\diamond\mathcal{P}_\ell$ and thus $\diamond\mathcal{P}$:

- if we have unbounded local memory (Section 7.4, [HW04]).
- if there is a *known* bound on the number of messages from the instable period (Section 7.5, [HW04]).
- if we have local clocks (Section 6, [BKM97]). Such algorithms are not message-driven.
- if we do not require self-stabilization [LLS03].
- if we drop the requirement for deterministic algorithms (Section 7.6, [HW05]).
- if the system has no timing uncertainty (Section 7.7, [HW05]).

Finally we showed that a local failure detector has the same computational power than a global one, since $\diamond\mathcal{P}_\ell$ can be transformed to $\diamond\mathcal{P}$ by a simple asynchronous self-stabilizing message-driven algorithm.

Chapter 8

Conclusion

In this thesis we considered the problem of failure detection under a sparsely connected network model. We addressed several aspects of such an approach:

By implementing the weakest failure detector for consensus under the weakest synchrony assumptions where this is possible for fully connected networks we are aware of, we showed that a sparse network is not more demanding in terms of synchrony that a fully connected one. Moreover, the complexity analysis showed a significant reduction in message load in sparse networks, when we consider a sparse network model for the algorithm, instead of simulating a fully connected network. One may argue that this approach impairs fault-tolerance and hence the achievable assumption coverage. However, if a neighbor is not timely, the messages of a non-neighbor routed over this process must in general be considered non-timely as well. Therefore, restricting the timeliness condition to links between neighbors is no loss of assumption coverage. Assuming a node degree larger the maximum number of crashed does not reduce the assumption coverage either, since if this requirement was violated, the fully connected overlay graph could also partition and therefore violate the system model. Moreover, a bound on the transmission delays of constant size messages between neighbors (without routing layer) can be established with very high coverage. This even yields an increase of assumption coverage compared to the original solution.

We also showed that a failure detector implementation can naturally profit from the implicit given network structure by reducing the message complexity of a failure detector algorithm when using this structure: We presented an implementation of an eventually perfect failure detector for a partitionable network with sparse topology. Between neighbors, an upper bound on the communication jitter is assumed. The number of neighbors is assumed to be bounded by Δ, which is an adequate model for wireless ad-hoc networks. The algorithm requires neither a priori knowledge of the number of processes in the system nor an upper bound on the communication delay between arbitrary processes. Every process broadcasts just $\Delta + 1$ messages per round to its neighbors, and under the assumption of a constant size name-space and time domain, these messages are of con-

stant size. Processes at shorter distances get more accurate information about each other than farther ones.

Finally, we considered the problem of self-stabilization. Whereas time-driven approaches are simple and can be implemented using only bounded memory, this is not the case for message-driven algorithms. We showed that there is no deterministic message-driven self-stabilizing implementation of $\diamond\mathcal{S}$ that uses only bounded memory if there is no bound on the channel capacity and there is a timing uncertainty. The result is tight in the sense that there are solutions if we drop one of the assumptions of the impossibility. We devised a simple solution, which requires unbounded memory—an assumption which is not reasonable when considering self-stabilizing algorithms for implementations in real systems. We therefore presented a practical solution which requires bounded memory. This solution can be used in real systems, as the crucial value M—a bound on the number of messages in the links—may be chosen in such a way that it is larger that the real number of messages in real systems. Further, we presented two other algorithms which show that it is possible to solve the problem by randomization and in system without timing uncertainty. It is thus possible to implement $\diamond\mathcal{P}_\ell$ and thus $\diamond\mathcal{P}$:

- if we have unbounded local memory,
- if there is a known bound on the number of messages from the instable period,
- if we have local clocks,
- if we do not require self-stabilization,
- if we drop the requirement for deterministic algorithms, or
- if the system has no timing uncertainty.

Further, we showed that a local failure detector has the same computational power than a global one, since $\diamond\mathcal{P}_\ell$ can be transformed to $\diamond\mathcal{P}$ by a simple asynchronous self-stabilizing message-driven algorithm.

All together, it seems beneficial to implement failure detectors directly on the sparse network instead of using a fully connected structure that is built atop of such a network by, e.g., routing.

List of Notations

$Q(p, q, t)$	set of messages which are in transit from p to q or vice versa at time t	23
$Q(p, q, t)$	set of messages which are in transit from p to q or vice versa after a step at time t	23
ε	jitter, $\varepsilon = \tau^+ - \tau^-$	25
Θ	communication delay uncertainty ratio, $\Theta = \tau^+/\tau^-$	25
$H(p, t)$	failure detector history	28
$\tilde{H}(p, t)$	inverse failure detector history	28
$\preceq_{\mathcal{E}}$	reducibility in environment $\mathcal{E}$	29
$\subseteq$	inclusion (relation between failure detectors)	29

Note that failure detector classes are not listed here.

Bibliography

[AB93] Yehuda Afek and Geoffrey M. Brown. Self-stabilization over unreliable communication media. *Distributed Computing*, 7:27–34, 1993.

[ACT99] Marcos Kawazoe Aguilera, Wei Chen, and Sam Toueg. Using the heartbeat failure detector for quiescent reliable communication and consensus in partitionable networks. *Theoretical Computer Science*, 220(1):3–30, June 1999.

[ACT00] 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.

[ADGFT01] Marcos Kawazoe Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Stable leader election. In *DISC '01: Proceedings of the 15th International Conference on Distributed Computing*, pages 108–122. Springer-Verlag, 2001.

[ADGFT03a] 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.

[ADGFT03b] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Solving leader election and consensus using one timely link. Technical report, LIAFA Universite D. Diderot, 2003.

[ADGFT04] Marcos K. Aguilera, Carole Delporte-Gallet, Hugues Fauconnier, and Sam Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 328–337, St. John's, Newfoundland, Canada, 2004. ACM Press.

[ADLS94] 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.

[AG93] A. Arora and M. Gouda. Closure and convergence: A foundation of fault-tolerant computing. *IEEE Transactions on Software Engineering*, 19(11):1015–1027, 1993.

[AH93] Efthymios Anagnostou and Vassos Hadzilacos. Tolerating transient and permanent failures (extended abstract). In *Proceedings of the 7th International Workshop on Distributed Algorithms (WDAG'93)*, volume 725 of *LNCS*, pages 174–188, Lausanne,Switzerland, Sept 1993.

[ALLT02] 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, Toulouse, France, Oct 2002. Springer Verlag.

[AT03] Marcos Kawazoe Aguilera and Sam Toueg. Failure detection and randomization: A hybrid approach to solve consensus. Technical report, Cornell University, 2003.

[ATD99] Marcos Kawazoe Aguilera, Sam Toueg, and Borislav Deianov. Revisiting the weakest failure detector for uniform reliable broadcast. In P. Jayanti, editor, *Distributed Computing: 13th International Symposium (DISC'99)*, volume 1693 of *Lecture Notes in Computer Science*, pages 19–34, Bratislava, Slovak Republic, sep 1999. Springer-Verlag GmbH.

[AW98] Hagit Attiya and Jennifer Welch. *Distributed Computing.* McGraw-Hill, 1998.

[BDM01] Özalp Babaoğlu, Renzo Davoli, and Alberto Montresor. Group communication in partitionable systems: Specification and algorithms. *Software Engineering*, 27(4):308–336, 2001.

[Bie03] 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.

[BKM97] Joffroy Beauquier and Synnöve Kekkonen-Moneta. Fault-tolerance and self-stabilization: Impossibility results and solutions using self-stabilizing

failure detectors. *International Journal of Systems Science*, 28(11):1177–1187, 1997.

[BMS02] 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.

[BMS03] Marin Bertier, Olivier Marin, and Pierre Sens. Performance analysis of a hierarchical failure detector. In *DSN*, pages 635–644, 2003.

[BO83] Michael Ben-Or. Another advantage of free choice: Completely asynchronous agreement protocols. In *Proceedings of the Second Annual ACM Symposium on Principles of Distributed Computing (PODC'83)*, pages 27–30, Canada, August 1983.

[CBGS00] Bernadette Charron-Bost, Rachid Guerraoui, and André Schiper. Synchronous system and perfect failure detector: solveability and efficiency issues. In *Proceedings of the International Conference on Dependable System and Networks (DSN'00)*. IEEE Computer Society Press, 2000.

[Cha90] Soma Chaudhuri. Agreement is harder than consensus: set consensus problems in totally asynchronous systems. In *PODC '90: Proceedings of the ninth annual ACM symposium on Principles of distributed computing*, pages 311–324, New York, NY, USA, 1990. ACM Press.

[Che00] Wei Chen. *On the Quality of Service of Failure Detectors*. PhD thesis, Cornell University, May 2000.

[CHT96] 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.

[CHTCB96] Tushar Deepak Chandra, Vassos Hadzilacos, Sam Toueg, and Bernadette Charron-Bost. On the impossibility of group membership. Technical Report 2782, Institute National de recherche en informatique et en automatique, January 1996.

[CKV01] Gregory V. Chockler, Idit Keidar, and Roman Vitenberg. Group communication specifications: A comprehensive study. *ACM Computing Surveys*, 33(4):1–43, December 2001.

[CM84] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Trans. Comput. Syst.*, 2(3):251–273, 1984.

[CT96] Tushar Deepak Chandra and Sam Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[CTA00] 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)*, New York City, USA, 2000.

[DDP03] 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.

[DDS87] 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.

[DFKM97] Danny Dolev, Roy Friedman, Idit Keidar, and Dahlia Malkhi. Failure detectors in omission failure environments. In *Proc. 16th ACM Symposium on Principles of Distributed Computing*, page 286, Santa Barbara, California, 1997.

[DGFF05] Carole Delporte-Gallet, Hugues Fauconnier, and Felix C. Freiling. Revisiting failure detection and consensus in omission failure environments. Aachener Informatik Berichte AIB-2005-13, RWTH Aachen, Department of Computer Science, 2005.

[DGG02] Assia Doudou, Benoit Garbinato, and Rachid Guerraoui. Encapsulating failure detection: From crash to byzantine failures. In *Reliable Software Technologies - Ada-Europe 2002*, LNCS 2361, pages 24–50, Vienna, Austria, June 2002. Springer.

[DGGS99] Assia Doudou, Benoît Garbinato, Rachid Guerraoui, and André Schiper. Muteness failure detectors: Specification and implementation. In *EDCC*, pages 71–87, 1999.

[Dij74] Edsger W. Dijkstra. Self-stabilizing systems in spite of distributed control. *Communications of the ACM*, 17(11):643–644, 1974.

[DIM97] Shlomi Dolev, Amos Israeli, and Shlomo Moran. Resource bounds for self-stabilizing message-driven protocols. *SIAM Journal on Computing*, 26(1):448–458, 1997.

[DLS88] Cynthia Dwork, Nancy Lynch, and Larry Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[Dol00] Shlomi Dolev. *Self-Stabilization*. MIT Press, 2000.

[DS98] 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)*, Puerto Vallarta, Mexico, 1998.

[FKM+95] Roy Friedman, Idit Keidar, Dalia Malki, Ken Birman, and Danny Dolev. Deciding in partitionable networks. Technical report, Cornell University, 1995.

[FLP85] Michael J. Fischer, Nancy A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM*, 32(2):374–382, April 1985.

[FS04a] Christof Fetzer and Ulrich Schmid. Brief announcement: On the possibility of consensus in asynchronous systems with finite average response times. In *Proceedings of the 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*, page 402, Boston, Massachusetts, 2004.

[FS04b] 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. (Brief announcement appeared at PODC'04).

[Gär02] Felix Gärtner. On crash failures and self-stabilization. Presentation at Journées Internationales sur l'auto-stabilisation, CIRM, Luminy, France, October 21-25, 2002, October 2002.

[GCG01] 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, Newport, RI, August 2001.

[GM91] Mohamed G. Gouda and Nicholas J. Multari. Stabilizing communication protocols. *IEEE Transactions on Computers*, 40(4):448–458, April 1991.

[GP01] Felix C. Gärtner and Stefan Pleisch. (im)possibilities of predicate detection in crash-affected systems. In A.K. Datta and T. Herman, editors, *Proc. Self-Stabilizing Systems : 5th International Workshop (WSS 2001)*,

volume 2194/2001 of *Lecture Notes in Computer Science*, page 98, Lisbon, Portugal, October 2001. Springer-Verlag.

[GP02] Felix C. Gärtner and Stefan Pleisch. Failure detection sequencers: Necessary and sufficient information about failures to solve predicate detection. In *DISC '02: Proceedings of the 16th International Conference on Distributed Computing*, pages 280–294. Springer-Verlag, 2002.

[GS96] R. Guerraoui and A. Schiper. Consensus service: A modular approach for building agreement protocols in distributed systems. In *Proceedings of the 26th IEEE Symposium on Fault Tolerant Computing Systems (FTCS-26)*, pages 168–177, 1996.

[Gue01] Rachid Guerraoui. On the hardness of failure-sensitive agreement problems. *Inf. Process. Lett.*, 79(2):99–104, 2001.

[Gue02] Rachid Guerraoui. Non-blocking atomic commit in asynchronous distributed systems with failure detectors. *Distributed Computing*, 15:17–25, 2002.

[HMR98] Michel Hurfin, Achour Mostefaoui, and Michel Raynal. Consensus in asynchronous systems where processes can crash and recover. In *Proceedings of the 17th IEEE Symposium on Reliable Distributed Systems*, pages 280–286. IEEE, oct 1998.

[HT93] 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.

[Hut04a] 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.

[Hut04b] Martin Hutle. On omega in sparse networks. In *Proc. 10th International Symposium Pacific Rim Dependable Computing (PRDC'04)*, Papeete, Tahiti, March 2004.

[HW04] Martin Hutle and Josef Widder. On the possibility and the impossibility of message-driven self-stabilizing failure detection. Research Report 34/2004, Technische Universität Wien, Institut für Technische Informatik, July 2004. http://www.ecs.tuwien.ac.at/projects/Theta/papers/.

[HW05] Martin Hutle and Josef Widder. Self-stabilizing failure detector algorithms. In *Proc. IASTED International Conference on Parallel and Distributed*

Computing and Networks (PDCN'05), Innsbruck, Austria, February 2005. Paper available at *http://www.ecs.tuwien.ac.at/projects/Theta/papers/*.

[KMMS03] Kim Potter Kihlstrom, Louise E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *Comput. J.*, 46(1):16–35, 2003.

[Lar02] Mikel Larrea. Brief announcement: Understanding perfect failure detectors. In *Proceedings of the 21st Annual ACM Symposium on Principles of Distributed Computing (PODC)*, page 257, Monterey, CA, July 2002.

[LFA99] 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, Bratislava, Slovaquia, September 1999. Springer.

[LFA01] Mikel Larrea, Antonio Fernández, and Sergio Arévalo. Eventually consistent failure detectors. In *SPAA*, pages 326–327, 2001.

[LFA02] 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)*, Gran Canaria Island, Spain, January 2002.

[LLS03] 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.

[MFVP05] Neeraj Mittal, Felix Freiling, Subbarayan Venkatesan, and Lucia Draque Penso. Efficient reductions for wait-free termination detection in crash-prone systems. Aachener Informatik Berichte 2005-12, RWTH Aachen, 2005.

[MMR03] 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.

[MR99a] Achour Mostéfaoui and Michel Raynal. Solving consensus using chandra-toueg's unreliable failure detectors: A general quorum-based approach. In P. Jayanti, editor, *Distributed Computing: 13th International Symposium*

(DISC'99), volume 1693 of *Lecture Notes in Computer Science*, pages 49–63, Bratislava, Slovak Republic, September 1999. Springer-Verlag GmbH.

[MR99b] Achour Mostéfaoui and Michel Raynal. Unreliable failure detectors with limited scope accuracy and an application to consensus. In *FSTTCS*, pages 329–340, 1999.

[MR00] Achour Mostéfaoui and Michel Raynal. k-set agreement with limited accuracy failure detectors. In *PODC '00: Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 143–152. ACM Press, 2000.

[MR01] Achour Mostéfaoui and Michel Raynal. Leader-based consensus. *Parallel Processing Letters*, 11(1):95–107, 2001.

[PS92] Stephen Ponzio and Ray Strong. Semisynchrony and real time. In *Proceedings of the 6th International Workshop on Distributed Algorithms (WDAG'92)*, pages 120–135, Haifa, Israel, November 1992.

[Ske81] Dale Skeen. Nonblocking commit protocols. In *SIGMOD '81: Proceedings of the 1981 ACM SIGMOD international conference on Management of data*, pages 133–142, New York, NY, USA, 1981. ACM Press.

[SM95] Laura S. Sabel and Keith Marzullo. Election vs. consensus in asynchronous systems. Technical report, Cornell University, Ithaca, NY, USA, 1995.

[SW05] Ulrich Schmid and Josef Widder. Achieving synchrony without clocks. (under submission), May 2005.

[Tel94] Gerard Tel. *Introduction to Distributed Algorithms*. Cambridge University Press, 1994.

[Völ04] Hagen Völzer. On randomization versus synchronization in distributed systems. Technical Report SIIM-TR-A-04-10, Universität zu Lübeck, June 2004.

[vR03] Jana van Greunen and Jan Rabaey. Lightweight time synchronization for sensor networks. In *Proceedings of the 2nd ACM international conference on Wireless sensor networks and applications*, pages 11–19, 2003.

[vRMH98] Robbert van Renesse, Yaron Minsky, and Mark Hayden. A gossip-style failure detection service. In *IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 98)*, 1998.

[Wid03] 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, Sorrento, Italy, October 2003. Springer Verlag.

[Wid04] Josef Widder. *Distributed Computing in the Presence of Bounded Asynchrony*. PhD thesis, Vienna University of Technology, Fakultät für Informatik, May 2004.

[WK03] Szu-Chi Wang and Sy-Yen Kuo. Communication strategies for heartbeat-style failure detectors in wireless ad hoc networks. In *DSN*, pages 361–, 2003.

[WLLS05] Josef Widder, Gérard Le Lann, and Ulrich Schmid. Failure detection with booting in partially synchronous systems. In *Proceedings of the 5th European Dependable Computing Conference (EDCC-5)*, volume 3463 of *LNCS*, pages 20–37, Budapest, Hungary, April 2005. Springer Verlag.

Curriculum Vitae

Personal

Name:	Martin Hutle
Title:	Dipl. Ing.
Born:	September 18th 1977, Feldkirch, Austria
Nationality:	Austrian Citizen
Familiy Status:	single
Parents:	Dorothea (maiden name: Schmidt) and Ditmar Hutle

Education

1983–1987	Elementary school, Volkschule Schwarzach
1987–1996	Secondary school, Bundesrealgymnasium Dornbirn. Passed the Matura with distinction.
1996–2002	Studying computer science at the Vienna University of Technology. Master thesis "Constraint Satisfaction Problems", performed at the Institute of Information Systems, Database and Artificial Intelligence Group. Received the master degree with distinction.
2000–	Studying Electrical Engineering at the Vienna University of Technology (branch: Communication Engineering).
2002–	Working as research assistant at the Institute for Automation and the Institute for Computer Engineering, Embedded Computing Systems Group; both at the Vienna University of Technology.