Diese Dissertation haben begutachtet:

_____ _____

DISSERTATION

Dynamic Aspects of Modelling Distributed Computations

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

UNIV.PROF. DR. ULRICH SCHMID

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

eingereicht an der Technischen Universität Wien
Fakultät für Informatik

von

DIPL.-ING. MARTIN BIELY

Matr.-Nr. 9526482

Leystrasse 110/1/14
A-1200 Wien
Europäische Union

Wien, 2009 _____

KURZFASSUNG

Diese Dissertation beschäftigt sich mit einer in der Regel vernachlässigten Facette der Modellierung verteilter Berechnungen: der Dynamizität. Die Folgen solcher, nämlich dynamischer, Annahmen werden hierbei anhand der Lösbarkeit des Konsensusproblems untersucht.

Klassischerweise werden die meisten Aspekte eines verteilten Systems als statisch angenommen; beispielsweise, gilt eine Komponente immerfort als fehlerhaft sobald sie sich einmal fehlerhaft verhalten hat. Ein anderer Aspekt für den dieses Dogma gilt, ist die Synchronität: Ab einem Zeitpunkt (oder auch von Beginn an) wird das System als für immer synchron angenommen. Im Gegensatz dazu beschäftigt sich diese Arbeit mit transienten Fehlern und vorübergehender Synchronität.

Der erste Teil der Arbeit stellt ein generelles Basismodell vor, welches in den folgenden Abschnitten in verschiedene Richtungen spezialisiert wird. Anschließend behandelt diese Arbeit die Beziehung zwischen verschiedenen Synchronitätsannahmen, wobei hier der Fokus zunächst nicht auf der Dynamizität liegt sondern auf die Verwandtschaften zwischen Modellen. Diese Verwandtschaften werden anhand von Modellen mit schließlich synchrone Subsystemen untersucht. Dabei zeigt sich, dass Algorithmen für solche Modelle ineinander transformiert werden können, wenn die schließlich synchronen Subsysteme gleich groß sind. Wie sich Dynamizität der Synchronität auf diese Verwandtschaften auswirkt wird anhand der Effizienz dieser Transformationen behandelt.

Der zweite und dritte Teil der Arbeit befasst sich mit dynamischen Fehlern. Zunächst wird für eine Klasse von dynamischen Kommunikationsfehlern gezeigt, dass es bei einer gewissen Anzahl solcher Fehler (in Relation zur Gesamtzahl der Prozesse im System) keine Lösung für das Konsensusproblem geben kann. Anschließend wird durch Design und Analyse optimaler Algorithmen gezeigt, dass diese Anzahl eine strenge Grenze darstellt. Im letzten Abschnitt des dritten Teils wird schließlich eine Lösung für das Konsensusproblem entworfen und analysiert, welche in Systemen mit nur vorübergehend verlässlicher und zeitgerechter Kommunikation (dynamische Synchronität) und vorübergehend byzantinischen Prozessen (dynamische Prozessfehler) korrekt funktioniert.

ABSTRACT

This thesis deals with an aspect of the modelling of distributed computations that is usually neglected in the existing literature: the possibly dynamic behaviour of the underlying system. The implications of employing assumptions that model this dynamic phenomena are investigated by analyzing their effect on the solvability of the consensus problem.

Classically all aspects of a distributed system are assumed to be static. For instance, a component that behaves faulty once is considered to be incorrect throughout the whole execution. Another aspect where this dogma is still valid (maybe to a lesser extent) is synchrony: Distributed systems are typically assumed to be synchronous forever (either from the beginning or from some point onwards). Contrasting these assumptions, this thesis is devoted to transient failures and intermittent synchrony

In the first part of the thesis a generic model is introduced which will be refined in several ways later on. Subsequently, the relations between certain kinds of synchrony assumptions are investigated. In doing so we do not focus on the dynamicity at first, but rather concentrate on the relation between models, which all have eventually synchronous subsystems (what this means exactly differs between the models). By using algorithm transformations, it is shown that all studied models are related which share the same size of their eventually synchronous subsystem. The efficiency of these transformations is finally used to reason about the relations between models where synchrony is only intermittent.

The second and third part of this thesis deals primarily with dynamic failures. First, it is shown that for a certain class of communication failures it is impossible to solve consensus if a certain ratio between the number of failures and the number of processes in the system is exceeded. Subsequently, it is shown through the design and analysis of optimal algorithms that this ratio is — in fact — a tight bound. Lastly, we devise an algorithm which is able to solve consensus in systems which alternate between synchronous and asynchronous periods (dynamic synchrony) and allows transient Byzantine faulty processes (dynamic failures).

CONTENTS

CHAPTER 1

INTRODUCTION

EVERY AREA OF THEORETICAL computer science has its pivotal questions and characteristic problems. For artificial intelligence it is the Turing test, for complexity theory it is the question of whether $P \stackrel{?}{=} NP$, in research on sequential algorithms it is sorting (or searching), etc. Although some of the above choices may be questionable, in the area of distributed computing the central problem definitely is "reaching agreement in the presence of faults" [Pease et al. 1980].

Distributed computing as a field is concerned with the understanding of the fundamentals and the formal treatment of distributed systems. In its most general sense a distributed system is a group of computing devices that work together without perfect knowledge of the global state. Working together requires that the devices can communicate with each other. With this very general definition in mind it should be no surprise that distributed systems are ubiquitous today: From a VLSI chip with its processing units, over a LAN with servers and workstations, to the Internet with everything connected to it, there are numerous examples of distributed systems.

Historically, the first problem related to distributed computing that received formal treatment was mutual exclusion[1] [Dijkstra 1965] and its self-stabilizing variant [Dijkstra 1974] in a shared memory system. In some sense, the latter also includes the notion of failures, as a self-stabilizing algorithm provides a way to recover from arbitrary failures: After all self-stabilizing algorithms provide a way to reach a set of "good states" from any initial state. The first formal treatment of a distributed message passing algorithm was by Le Lann [1977], who introduced the leader election problem. The coordinated attack problem [Gray 1978] seems to be the first problem in which fault-tolerance played a central role. The purpose of the coordinated attack problem was to model the commit operation in distributed databases. Along with its definition Gray also presented a proof showing that it is impossible to solve even in a system with only two processes when communication is unreliable. (As one can see proofs of impossibility have always been an aspect central to distributed computing.)

Contrasting this impossibility result, both the coordinated attack problem and — our main focus — the consensus problem are rather trivially solved in systems without failures. Given a set of processes each with a local input value, the aim of consensus is to

[1] Starvation was possible with his solution, however, as pointed out by Knuth [1966].

locally calculate an output value that is the same at all processes. The input value is often called initial value, or proposal, while the output value is referred to by the decision value. In the absence of failures one could simply let all processes broadcast their proposals and let them then decide on the locally determined minimum or maximum of all received values. Another would be to take the most common value (breaking ties may be necessary). In failure-prone systems such simple approaches will not work as processes may see different sets of proposals. However, failures cannot be ignored, since fault-tolerance is one driving force for employing distributed systems in the first place. The requirement for fault-tolerance can be addressed by replicating the (failure-prone) components. For these replicas to be useful, one has to ensure that the data they operate on are always consistent, which in turn leads to the need to solve fault tolerant consensus. Regarding fault-tolerance, research on the consensus problem has concentrated on static process failures since its beginning [Lamport et al. 1982] (see [Fischer 1983] for an early survey). This might have been caused by the aforementioned negative result on coordinated attack, which is closely related to consensus.

Before discussing the consensus problem itself in more depth, we first shed some light on the system assumptions that influence the definition of the problem and its solvability.

1.1. Failures

As mentioned before, one important aspect of solving consensus is the set of assumptions made about the behaviour of faulty components. Classically, there are two dimensions to this aspect: one is which components may fail, the other is how they behave when they do fail.

Regarding the first aspect, we have already seen examples above: On the one hand we have Gray's [1978] impossibility, which is an example where the communication system (or parts thereof) may fail. On the other hand, we have already noted that the focus of research on consensus was on systems where processes may fail. Models in which both links between processes and the processes themselves may fail would be "more realistic, [but] little known and seldom used." [Santoro 2007, page 411]

With respect to the second aspect the most general assumption is that the failed components may behave arbitrarily. Failed components may even maliciously cooperate to deceive the non-faulty components. This kind of behaviour has been termed Byzantine [Lamport et al. 1982]. Clearly, an algorithm should be designed in a way such that even maliciously cooperating processes cannot fool another process into unspecified behaviour. Thus, processes have to be cautious in how they use data received from other processes. This is contrasted by benign faulty components, whose output can be trusted but may be missing. In the context of benign failures, we differentiate between crash and omissions. A crashed component remains mute from some point in time onwards, whereas an omissive component is only mute from time to time.

It is obvious that arbitrary behaviour includes benign behaviour, and that a component that may exhibit any omission pattern can also behave in the same way as a crashed component. The reason why it is not always assumed that faulty components can behave Byzantine is due to different requirements regarding the number of processes vs. the number of failures. If we consider process failures only, then it is impossible to solve consensus when more than a third of the processes may fail [Lamport et al. 1982]. When process only crash (or are omissive) consensus remains solvable, even when all but one

process may fail, when the system is synchronous enough. This naturally leads to the question of how many processes are required when some processes may behave Byzantine while others may only crash. There is a wide range of literature on models aimed at answering this question for consensus (e.g., [Meyer and Pradhan 1987; Thambidurai and Park 1988; Lincoln and Rushby 1993; Walter and Suri 2002; Weiss and Schmid 2001; Schmid et al. 2002; Biely 2003; Biely et al. 2009; Schmid et al. 2009]) as well as other problems (e.g., [Cristian and Fetzer 1994; Rushby 1994; Walter et al. 1994; Azadmanesh and Kieckhafer 1996; Schmid 2000; Siu et al. 1998a; Schmid and Schossmaier 2001; Azadmanesh and Kieckhafer 2000; Widder and Schmid 2007]). Collectively these model (and others which consider different failure behaviours) are often referred to by the term hybrid failure models. Santoro [2007] uses the term only for models which consider both process and link failures.

In fact, there are only a few failure models for synchronous systems in the literature that deal with link failures explicitly at all (we will discuss them here only shortly, see Chapter 7 for a more detailed discussion). One argument often used against explicitly modelling link failures is that one could simply map link failures to process failures (as it is done by, e.g., Gong et al. [1995] and [Perry and Toueg 1986]). However, for significant link failure rates, this quickly leads to the exhaustion of the maximum number f of tolerable process failures [Schmid et al. 2009]. Another class of models [Sayeed et al. 1995; Siu et al. 1998a] considers a small number of link failures explicitly, but as at most $\mathcal{O}(n)$ links are assumed to be faulty system-wide during the entire execution of a consensus algorithm in these approaches, they can only be applied in case of very small link failure rates. There are only two approaches which are applicable to synchronous systems with larger link failure rates: The work by Pinter and Shinahr [1985] and the perception based failure model (e.g., [Schmid 2000; Weiss and Schmid 2001; Schmid et al. 2002; Biely 2003; Widder and Schmid 2007; Biely et al. 2009; Schmid et al. 2009]), with the latter being an example of a (hybrid) model which considers different classes of failures for both processes and links. The key to tolerating more than just $\mathcal{O}(n)$ total link failures is to allow different links to fail in different rounds, which leads to the third dimension of failures.

We now turn our attention to the third dimension of failures: the dynamics of failures. Unfortunately, dynamic failures are somewhat neglected in research with the exceptions (apart from the aforementioned two approaches) including Santoro and Widmayer [1989] and Charron-Bost and Schiper [2006a; 2009]. Currently, the major textbooks on distributed computing ([Tel 1994; Lynch 1996; Attiya and Welch 2004]) do not mention dynamic failures, with Santoro's book [2007] being an obvious exception to this rule. The reason for this situation is a dogma which emerged from the fact that "the classical models of fault tolerant distributed systems only handle faults that are *static both in space and time*, i.e., faults by an unknown but static set of (so-called faulty) processes (static faults in space) that are considered to be faulty for the whole computation (static faults in time)"[Charron-Bost and Schiper 2007]. Ignoring this dogma one can consider failures to be dynamic in time (transient) and space (mobile).

One approach to model such dynamic failures is by considering transmission faults — an approach pioneered by Santoro and Widmayer. Moreover, transmission faults can represent the failure of links as well as processes [Santoro and Widmayer 1989] — even permanent faults. As an example, consider a process p has crashed: the other processes perceive (only) that there are no more transmissions from that process (from some point in time on). When considering a Byzantine faulty process, it can be modelled as that

process behaving correctly but all its outgoing transmissions are changed such that it will appear as behaving Byzantine from the outside. This is the approach taken in the first half of Chapter 6 (cf. [Biely et al. 2007a]).

When processes can recover from behaving Byzantine, this approach cannot be taken with agreement problems, since recovery usually involves returning to a predefined recovery state, otherwise it cannot be (reasonably) guaranteed that the decision values are related to the proposed values. More generally, this applies to all problems which follow a similar input-output pattern. Other problems, such as clock-synchronization, where there are no input values that may be modified/lost in phases of Byzantine behaviour, it is not necessary to "mark" the process as recovering by entering a special state [Barak et al. 2000; Anceaume et al. 2004b; Anceaume et al. 2007].

Anyhow, there are (in essence) two approaches for modelling systems where all processes may behave Byzantine: (i) self-stabilizing systems and (ii) systems with bounded failure rates.

Bounded failure rates are usually modelled by bounding the number/fraction of processes which may be faulty in some sliding time window. This approach has been taken to provide perpetually synchronized clocks [Anceaume et al. 2007; Anceaume et al. 2004a; Barak et al. 2000]. In a self-stabilizing system, all processes may be faulty at the same time, but properties can only be guaranteed when all processes have returned to normal behaviour. This is modeled by starting the processes from an arbitrary state. More recent research also includes the possibility of some processes being faulty after stabilization as well (this kind of assumption is often indicated by usage of the term "self-stabilizing fault-tolerance"). Self-stabilization has been used to analyze a wide range of problems, starting from mutual exclusion [Dijkstra 1974] to clock synchronization [Dolev and Welch 2004; Hoch et al. 2006]. A variant of consensus was considered by Dolev et al. [2006], where consecutive instances of consensus were executed to drive a self-stabilizing replicated state machine [Schneider 1990]. Another variant of consensus which is sometimes considered, is one where the input values are not fixed at the beginning of the execution, but may change arbitrarily often. With this variant (called stabilizing consensus) processes may decide multiple times but are required to eventually stabilize their output, when the inputs have stabilized (e.g., [Angluin et al. 2006; Kutten and Masuzawa 2007]).

1.2. Synchrony

When there are no failures, consensus can be solved even when there is no information on the timing of the system. That is, all delays (times between events) in the system can be arbitrarily large (as long as they are finite). That is, the links may take as long as they like until messages are delivered to their receivers, and processes may take as long as they like to respond to messages they receive. But when there is the possibility of a single crash, consensus becomes impossible to solve [Fischer et al. 1985]. The problem is, that processes can never know whether some message from another process will eventually arrive or whether the other process has stopped taking steps forever before sending it. When communication and processing speeds are arbitrary then the system is referred to as being asynchronous.

On the other end of the spectrum, one finds the assumption that the (relative) speed of processing as well as a bound on the communication delays are known. Such systems are called synchronous. With this knowledge one can build the abstraction of the perfectly

synchronous rounds. Here, the distributed computation proceeds in rounds, with every process sending their messages, and receiving all the messages the other processes have sent in the current round, and finally doing some computation (including preparing the message to be sent in the next round). While very powerful with respect to the easy solvability of problems such strong assumptions are often "impossible or inefficient to [guarantee] in many types of distributed systems" [Lynch 1996, page 5].

These two extremes and their fundamentally different results regarding the solvability of consensus — [Fischer et al. 1985] vs. [Lamport et al. 1982] have inspired research into what lies between these extremes, leading to contemplating systems with different degrees of synchrony. Classically synchrony is captured by bounds on the two aforementioned timings: the message delays and the relative computing speeds. Dwork, Lynch, and Stockmeyer [1988] showed that it is also possible to solve consensus in the presence of Byzantine processes when bounds on these exist but only hold eventually. Moreover, they have shown that it does not matter for solving consensus whether one set of parameters is assumed to hold for all executions (informally speaking the values of the parameters are known) or whether the parameters take different values for each execution (informally speaking the values of the parameters are unknown).(In Chapter 3 we will explore this question more generally and come to the conclusions that there is no difference for the solvability of distributed computing problems in general.) Besides synchrony, the effect of other model parameters (such as atomicity of send and receive steps, or message ordering on links) on the solvability of distributed computing problems has also been examined by Dolev et al. [1987].

Although partially synchronous models typically also assume the existence of bounds on communication and processing speeds, the key difference to the synchronous system is that there is some uncertainty, which either stems from the absence of bounds which hold for all executions or from the possibility that these bounds only hold after some finite time. In the classic partial synchronous model [Dwork et al. 1988] the communication delay is modelled by how many steps processes can take before a message has to be delivered. The relative computing speeds of processes is expressed as the number of steps processes can take such that every alive process has taken at least one step during the same time interval.

One important difference between these three classes of models (asynchronous, synchronous and partially synchronous) can be observed when considering systems with reliable communication and crash failures: In an asynchronous system, a process can never tell whether another process has crashed (or is only slow), while in synchronous round based models a process knows when another process has crashed at the end of the round (or more generally after a fixed timeout). In partially synchronous systems, it is only eventually that a process can detect other processes crashes. However, it can never be sure that this detection is correct, since the detection might originate in its estimates of the unknown bounds being wrong, or the time after which the bounds hold still being in the future. This is what is essentially captured by the concept of unreliable failure detectors [Chandra and Toueg 1996], which are commonly thought of as a possibility to abstract from the particular timing assumptions in a system. This view is based on a misunderstanding, however, failure detectors do not provide an abstract view of time [Charron-Bost et al. 2000]. In fact, one could consider the information that failure detectors provide as too abstract: Chandra et al. [1996] show that Ω the eventual leader election oracle is the weakest failure detector to solve consensus. Their result does not

provide any information on the weakest assumptions to solve consensus however, as the definition of Ω and any other (eventual) failure detector necessarily includes a property that has to (eventually) hold forever (cf. [Charron-Bost et al. 2008]). As we will see in Chapter 3 "eventual forever" properties of failure detectors are equivalent to "eventual forever" synchrony assumptions. For solving consensus, such assumptions are superfluous however, as it is long known that it is sufficient for the system to be synchronous for some time [Dwork et al. 1988]. That is, failure detectors enforce static synchrony properties, although dynamic synchrony assumptions would be sufficient. Besides pointing out this fact Charron-Bost et al. [2008] demonstrate, that the meaning "at least as strong" usually given to the classic relation between failure detectors [Chandra and Toueg 1996] does not match its properties, because there is a failure detector that is not as strong as itself. These realizations (especially the first one) makes the chase for the weakest model for implementing Ω [Hutle et al. 2009] somewhat paradoxical, as it is widely — but wrongly nonetheless — assumed to lead to the weakest model for solving consensus.

One promising possibility to abstract from the particular timing assumption is the use round-based models (not to be confused with the lock-step synchronous round model), which are solely based on what the processes are guaranteed to perceive within one round. This is the underlying idea of the work by Santoro and Widmayer [1989], Gafni's [1998] round by round failure detectors, the HO-Model (e.g., [Charron-Bost and Schiper 2006a; Charron-Bost and Schiper 2006b; Biely et al. 2007a; Charron-Bost and Schiper 2009]), and the GIRAF-framework [Keidar and Shraer 2006; Keidar and Shraer 2007]. With respect to dynamic synchrony the advantage of round-based models is that one can quite simply define intermittent synchrony (not all aforementioned approaches do so, however). One simply assumes guarantees about the perceptions to hold for some rounds only. Indeed, when determining the minimal duration of the synchronous period Dwork et al. [1988] take a two step approach by determining the number of rounds and then determining the time it takes to execute these rounds. However, not all algorithms require the all "good" rounds to occur in one block. One example where this is not the case is the $\mathcal{A}_{T,E}$ algorithm [Biely et al. 2007a]. When implementing such round abstractions one does not have to assume a single (possibly quite long) synchronous period, but can rather settle for systems that alternate between stable (i.e., synchronous) and unstable (i.e., synchrony assumptions do not hold) periods (cf. Chapter 6 or [Hutle and Schiper 2007a]). The problem of reaching agreement in systems with such alternating periods directly (i.e., without constructing a round abstraction) is considered by Lamport [1998] in his Paxos paper. A wider range of problems was analyzed in the timed asynchronous model Cristian and Fetzer [1999] which also features alternating stable/unstable periods.

1.3. Atomicity

In distributed computing research, computations are usually not assumed to be continuous. Rather computations are split into steps, which is partly due to the fact that distributed computations need input from other processes, and might have to wait for that input. These steps are almost always assumed to be atomic, that is, either the whole step (which might include multiple operations) is performed as a whole, or not at all.

Synchrony is related to when processes may take these steps. Atomicity is concerned with the orthogonal issue of what may be done (atomically) within one step. Not too surprisingly, there is a wealth of different atomicity assumptions in the existing literature.

What is common to all these assumptions, is that steps only occur when a process sends or receives a message. Thus computation without interaction with the environment, that is, the complexity of the computation, is assumed to have no influence on the time it takes for a distributed algorithm to be executed. This is reflected by the fact that typically (with one exception [Moser and Schmid 2006; Moser 2009]) these steps have zero duration, which also has the advantage of being able to abstract away nearly all queueing effects.

Dwork et al. [1988] assume that a process can either receive some deliverable messages (that is an arbitrarily but finite large set) or send a single message within one step. Fischer et al. [1985] allow the reception of a single message and the sending of a finite (possibly empty) set of messages within a single atomic step. Failure detector based models [Chandra and Toueg 1996] modify the latter by assuming that in one atomic step a process can receive a message (if one is available), query its failure detector, and may send messages. Dolev et al. [1987] have shown that there is only a difference with respect to the solvability of consensus between send-and-receive-only and send+receive steps when communication is synchronous but process are asynchronous (i.e., there is no bound on the relative processing speed). Note that the ability to receive more than one message in a step in the model by Dwork et al. [1988] is necessary in order to be able to define a bound on the message transmission delay in terms of the number of steps taken. (Otherwise, queuing effects could lead to any bound being violated.)

In all the models discussed above the computation is driven by the progress of time, that is processes can take steps by themselves (or can be seen as being triggered by timers). Thus such models are referred to as being time-driven. The alternative are message-driven systems where steps are only triggered by the reception of messages, and thus there is no choice but to assume that the reception of exactly one message triggers the sending of a (finite possibly empty) set of messages in a single step (cf. for example [Robinson and Schmid 2008]).

1.4. Consensus

After discussing different model parameters (some of which influence the problem definition), we now return to the consensus problem itself by defining the problem: Each process p_i has an input value x_i (also called proposal) from some set of possible values $\mathcal{V}$ and has to irrevocably decide on an output value, that is, it has to determine an output value y_i. There are some restrictions on legitimate decision values, which come by the names of Termination, Validity and Agreement:

Termination requires each non-faulty process to eventually decide. For synchronous systems the time after which all non-faulty processes have decided can even be bounded, for instance by stating the number of rounds the algorithm takes. Note that when up to t processes may be faulty, any algorithm has to take at least $t + 1$ rounds. (Fischer and Lynch [1982] have shown this for the Byzantine failure case, Merritt [1985] based on ideas from Dolev and Strong [1982b] for the crash failure case.) For partially synchronous and asynchronous systems the execution time of algorithms cannot be bounded.

Validity is needed to rule out trivial solutions that always decide on the same value, say 42. We have to distinguish two variants of this property depending on whether one considers (i) Byzantine failures or (ii) only benign ones: For (ii), the Validity requirement is that processes only decide on the initial value of some process. Since it is undesirable to decide on a value that a Byzantine process proposes, the variant for case (i) requires

that if all non-faulty processes share the same input value, then this is the only possible decision value.

Agreement classically demands that the decision values of two non-faulty processes are equal. Note that this definition allows for faulty processes to decide any value. While this is desirable for Byzantine faulty processes, it gives rise to a highly undesirable phenomenon: Since processes are only considered non-faulty when they never behave erroneous (e.g., by crashing), "a faulty process is allowed to decide differently from the non-faulty processes even it crashes a very long time after making a decision" [Charron-Bost and Schiper 2004]. When consensus is viewed in isolation this possibility is highly counter intuitive, but when one considers larger systems where consensus is only a building block the situation becomes worse, as the local inconsistency at the faulty process might contaminate larger parts of the system [Gopal 1992]. This cannot happen with the uniform variant of the Agreement property (termed total consistency in [Dwork and Skeen 1984]), which requires that no two processes decide on different values. The variant of consensus that ensures Uniform Agreement instead of Agreement is called uniform consensus. It has been shown by Charron-Bost and Schiper [2004] that in synchronous systems the uniform variant is harder. The difference lies in a larger minimum number of rounds required before processes can decide early. Early decision refers to the possibility to decide before round $t + 1$ if less than t processes crash. In contrast, Guerraoui [1995] has shown that for those partially synchronous systems which can be characterized by eventually reliable failure detectors [Chandra and Toueg 1996] there is no difference between uniform and non-uniform consensus. In Chapter 3, we show that for large classes of partially synchronous models there is no difference in terms of solvability in general.

The agreement and validity properties are sometimes also referred to as consistency and unanimity (or integrity), respectively. Together they ensure the safety of a consensus algorithm. Note that both properties are necessary for the problem to be non-trivial: On the one hand one could decide that agreement is not necessary, then any process can decide on its input value, and validity is ensured. On the other hand when one omits validity, all processes can always decide on the same fixed value independent of any of the input values, thereby ensuring agreement.

1.5. Outlook

This thesis is split into three major parts: The first part starts with the introduction of a generic model of distributed computation (Chapter 2) which will be used as a framework for the definition of different models in later sections.

Then we explore the relations between models in Chapter 3: Here the aim is to determine whether different partially synchronous models are equivalent regarding solvability. We can answer this question in the affirmative, which contrasts the earlier work of of Charron-Bost et al. [2000], who have found a difference with this respect between two synchronous model. This difference was illustrated by presenting a problem which is solvable in a model based on timing assumptions (synchronous processes and synchronous communication in the notations of Dolev et al. [1987]) but not in the asynchronous system augmented with the perfect failure detector. Among other things we show in Chapter 3 that it is possible to solve the same problems in the "eventual variants" of these two models. To this end we introduce the notion of algorithm transformations. We also analyze the cost of these transformations, that is the cost of ensuring the

assumptions of one model based on the properties of another model.

The second part (Chapter 4) is devoted to proving an unified impossibility result for Byzantine process and dynamic link failures. In the third part (Chapter 5) we then complement this result by showing how to mask certain types of dynamic link failures (in the presence of static process failures). What is particularly interesting about the tight lower bound (which is provided by impossibility result and the matching possibility result) is that it shows for the (to the best of our knowledge) first time that lower bounds are not necessarily additive. While parts of the impossibility result were known before (from the work of Lamport et al. [1982] and Fischer et al. [1986], respectively, Schmid et al. [2009]), we provide a novel combination of these results using a chain of proofs, which are all based on the same proof technique. The resulting impossibility result does not depend on the link failures being dynamic, while for the possibility result we mask dynamic link failures. For this possibility result we employ a round model which is based on the assumption of perpetual synchrony (although, arguably, the (intermittent) omissions on some links could also be caused by (intermittent) lack of synchrony in the communication).

We then — in Chapter 6 — turn our attention to models where synchrony is only intermittent as well. Here, we start from a round model (the aforementioned HO-model), as the high level abstraction of round models provide a convenient way to define intermittent synchrony. Regarding failures we naturally follow the approach of considering only transmission failures [Santoro and Widmayer 1989] as it is typically done in the HO-model. In contrast to the work by Charron-Bost and Schiper [2009] we also allow transmissions to be value faulty. We present an algorithm that is derived from the OneThirdRule algorithm [Charron-Bost and Schiper 2009] by replacing the two thresholds in the algorithm by parameters. We then derive predicates over HO-sets and conditions for relations of the parameters for which safety and liveness of the algorithm can be guaranteed. Based on these results we then implement a round structure around this algorithm which ensures these predicates in a system with intermittent synchrony and intermittent Byzantine processes. The resulting algorithm is safe as long as at most a fifth of the processes are faulty at any point in time (and faulty processes do not change to fast) and is live (i.e., solves consensus) when there is some time when no process is faulty. When no process is faulty it is also fast, that is, it decides within one round.

Finally, in Chapter 7 the results are summarized and we conclude with additional discussions of related approaches.

Some of the results presented in this thesis have in part been previously published [Biely et al. 2007a; Biely et al. 2007b; Biely and Hutle 2009] or are currently under submission [Biely 2006; Biely et al. 2009].

PART

ON MODELS

CHAPTER 2

A GENERIC MODEL FOR DISTRIBUTED COMPUTATIONS

AS WE HAVE NOTED in the introduction, different models with different assumptions are used in the literature on distributed computing. In this chapter, we provide a model which allows us to unify a large number of these models. Different instances of this model adapted to particular needs will be used in the subsequent chapters. This approach allows us to better compare the different results to each other and to the existing literature.

One important aspect of models in distributed computing is that of how the processes and the connecting network is represented, another how the computation itself is modelled. Modelling the former is pretty standard: We represent a distributed system by a graph $\langle \Pi, \Lambda \rangle$, where Π is the set of nodes and Λ is the set of directed edges between pairs of nodes. To each node we associate a process (defined below) and to each edge we associate a directed link (also defined below) connecting the processes associated with the edge's endpoints. Thus, a bidirectional (network) link is represented by a pair of edges. We denote by $\langle p, q \rangle$ the channel from process p to q. To simplify the presentation, we assume that every process has a link to itself ($\forall p \in \Pi : \langle p, p \rangle \in \Lambda$). This allows us to model the complete network (one special graph which we consider) via $\langle \Pi, \Pi \times \Pi \rangle$. The number of processes in the system is abbreviated by n, i.e., $n := |\Pi|$.

2.1. Processes

Before we turn to the aspect of how to model distributed computations, we examine how computations are modelled in the uni-processor case. One possibility that is frequently used are state machines. A state machine is defined by a (possibly infinite) set of states $\mathcal{S}$, a subset of those being the set of initial states $\mathcal{I}$, and a relation that defines state transitions $\mathcal{N}$. There are a number of ways to define these transitions. We follow what Lamport [2008] calls state-action behaviour, in this case $\mathcal{N} \subseteq \mathcal{S} \times \mathcal{A} \times \mathcal{S}$, where $\mathcal{A}$ is the set of all actions. A computation is then defined by a (possibly infinite) alternating sequence

of states from $\mathcal{S}$ and triples $\langle s, \triangleright, t \rangle$ taken from $\mathcal{N}$ (that is, s and t are states and $\triangleright$ is an action). However, not every possible sequence of such triples also characterizes a legal computation, which leads to the notions of applicability and admissibility.

An action $\triangleright$ is applicable to some state s, if there is some t such that $\langle s, \triangleright, t \rangle \in \mathcal{N}$. The state after applying action $\triangleright$ to s is the state t. Thus one can write the aforementioned sequence more conveniently as $state^0, \triangleright^0, state^1, \triangleright^1, state^2, \ldots$. This sequence, however, only adequately describes a computation, if $state^0 \in \mathcal{I}$. In this case this is the only admissibility condition and we call executions for which it holds admissible.

To model a distributed computation, we use essentially the same approach, albeit, there is not one state machine, but there are multiple state machines interacting through some means of communication. We subsequently also model the communication by state machines,[1] which somewhat complicates the conditions we get for the applicability of actions. For instance, it is clear that, when a process changes its state based on a message it has received, then this message must have arrived for the state change to be applicable. Moreover, in many distributed computing models, the notions of failures and time (which do not exist in other fields of computer science which normally use state machines) dictate more complicated conditions for admissibility.

The modelling of the interactions of a process with its incident links (and other parts of the environment) is based on the actions, respectively, on additional information the actions carry. In order to define an execution of a distributed algorithm, we then combine all the state machines representing processes with those that represent their environment (that is the state machines that represent the links between the processes). We use admissibility conditions to ensure that the one (big) state machine we obtain by the composition is one whose state transitions model the execution of a distributed algorithm.

In accordance with these discussions, we define each process p to be a state machine with the set of states $\mathcal{S}_p$, the set of initial states $\mathcal{I}_p$, and the next-state relation $\mathcal{N}_p \subseteq \mathcal{S}_p \times \mathcal{A}_p \times \mathcal{S}_p$, as for the non-distributed case above. What is new, is that we give additional semantics to the actions (in the set $\mathcal{A}_p$).

Next we introduce the notion of an execution restricted to a process: An execution of p's state machine, denoted $\varepsilon|_p$, is an alternating sequence of states of p and actions of p. We again restrict the set of all possible sequences by demanding that each action must be applicable to the state before it — what it means to be applicable may depend on the kind of action — and that the state following it is specified by the next-step relation $\mathcal{N}_p$. That is, for any execution of p, $\varepsilon|_p = state_p^0, \triangleright^0, state_p^1, \ldots$, it must hold that:

1. $state_p^0 \in \mathcal{I}_p$,
2. $\forall i \in \mathbb{N} : \langle state_p^i, \triangleright^i, state_p \rangle \in \mathcal{N}_p$, and
3. $\triangleright^i$ is applicable to $state_p^i$.

In order to model the information exchange between the links and the processes we associate with each incoming link ($\langle q, p \rangle \in \Lambda$) of a process p the set $I_{\langle q,p \rangle}$ representing the in-buffer, which is part of $state_p$. These sets store the messages taken from the set of all messages $\mathcal{M}$, p has received from q. Thus $\mathcal{S}_p = \mathcal{L} \times \mathcal{M} \times \ldots$, with $\mathcal{L}$ denoting the set

[1] In this, our approach is similar to the one taken by Lynch [1996; Kaynar et al. [2006], who use timed I/O automatons for both processes and links.

of possible local states of p. Given some state $state_p$, we define by $I_{\langle q,p\rangle}(state_p)$ the value of the in-buffer $I_{\langle q,p\rangle}$ in $state_p$ and by $local(state_p)$ the local part of $state_p$.

The first class of state transitions we look at are transitions that model the process performing a local computation. For these transitions there is only one action $\triangleright_c = \langle compute\rangle$. As defined above (in the uni-processor case) a transition $\langle s, \langle compute\rangle, s'\rangle \in \mathcal{N}_p$ is applicable only in state s. Moreover, such transitions may only modify the local state of p, that is for some $\langle s, \langle compute\rangle, s'\rangle$ it must hold $I_{\langle q,p\rangle}(s) = I_{\langle q,p\rangle}(s')$ for all $\langle p, q\rangle \in \Lambda$, and may also not depend on the contents of any set I.

For message-passing distributed algorithms the most important additional kinds of actions are those that allow a process to interact with the communication subsystem. The information exchange between the links and the processes, is done via a number of actions and the aforementioned buffers. Process p sending a message (taken from the set of message $\mathcal{M}$) to a process q is modelled by a sending action, denoted $\triangleright_s = \langle send, m, q\rangle$, with $m \in \mathcal{M}$ and $\langle p, q\rangle \in \Lambda$. A sending action $\triangleright_s = \langle send, m, q\rangle$ is applicable in a state $state_p$ when there exists a triple $\langle state_p, \triangleright_s, state'_p\rangle \in \mathcal{N}_p$. When such a state transition is applied to $state_p$, only the local state of the machine may change (to $local(state'_p)$), while all I sets remain unchanged—just like in the case of computation action transitions. A receiving action $\triangleright_r = \langle recv, m, q\rangle$ is applicable in a state $state_p$ when there exists a triple $\langle state_p, \triangleright_r, state'_p\rangle \in \mathcal{N}_p$ and m is contained in the set $I_{\langle q,p\rangle}(state_p)$. When this state transition is applied to $state_p$, in addition to the local state transition from $local(state_p)$ to $local(state'_p)$, the message m is removed from the in-buffer, that is, $I_{\langle p,q\rangle}(state'_p) = I_{\langle p,q\rangle}(state_p) \setminus \{m\}$. All other in-buffers remain unchanged, i.e., $I_{\langle p,q\rangle}(state'_p) = I_{\langle p,q\rangle}(state_p)$. Thus a receiving action models the case where a process looks into its in-buffers (all at the same time) and takes note of one of the messages present.[2] Obviously, all receive buffers could also be empty, in this case the (special) receiving action $\triangleright_r = \langle recv, \bot, \bot\rangle$ is applicable in $state_p$ if $\langle state_p, \langle recv, \bot, \bot\rangle, state'_p\rangle \in \mathcal{N}_p$ for some $state'_p$ and we demand, that whenever any other receive action is allowed (via $\mathcal{N}_p$) to occur in $state$, then so must this one. Moreover, $\bot$ may not belong to the set of messages $\mathcal{M}$.

In order to explain how messages are added to the set $I_{\langle \_\_,p\rangle}$ in the first place, we introduce deliver actions $\triangleright_d = \langle deliver, m, q\rangle$ as an additional action type which may occur in $\varepsilon|_p$. Deliver actions are applicable independent of the state of the process p, i.e., for every possible deliver action $\triangleright_d$ we have $\forall s \in \mathcal{S}_p : \langle s, \triangleright, s'\rangle \in \mathcal{N}_p$. The result of these actions is simply to add m to $I_{\langle p,q\rangle}(s)$ and leave all other components of $state_p$ unchanged.

Apart from communication with other processes, some models also assume that processes have additional means of obtaining information about the (global) execution. Most commonly in literature, are the ability of processes to read clocks or failure detectors. We model both by allowing the processes to read "special" variables through reading actions $\triangleright = \langle read, variable, value\rangle$. Consider the example of a clock, then $variable$ would be $clock$ and $value$ the clock value read. Clearly, not every possible clock value is a legal choice to be read at any point of the execution (e.g., clocks are usually assumed to be monotonically increasing in their value). We do not enforce such constraints on this level, but leave it to the specific model to define adequate admissibility conditions. Likewise

[2] In this thesis, we assume that any message can be received by a receiving action. That is, any combination of message and process identifier that a process may find in its in-buffer must be applicable anytime a receiving action is performed. However, it is also possible for a model to allow an algorithm to restrict the set of messages applicable at some point via the next state relation.

we also define an additional output action, which involves assigning a value to a "special" variable. (As an example where this is useful, consider consensus where deciding can be seen as setting the decision variable to the decision value.) To this end we also allow actions of the form $\triangleright = \langle \mathit{write}, \mathtt{variable}, \mathtt{value} \rangle$. With respect to applicability, both are applicable in the same way as computation actions. Which variables exist and what their domains are, depends on the particular model (or problem), which may also specify additional restrictions on the $\mathtt{value}$ (such as the assumption that it is monotonically increasing in case of clocks) through admissibility conditions (or correctness conditions, in case of problems).

2.2. Links

We now turn to the second kind of component in a distributed system, that is, links. In our modelling the link $\langle p, q \rangle \in \Lambda$ is a state machine defined by $\mathcal{S}_{\langle p,q \rangle}$, $\mathcal{I}_{\langle p,q \rangle}$, and $\mathcal{N}_{\langle p,q \rangle}$ as above, as well as $\mathcal{A}_{\langle p,q \rangle} = \{\triangleright_s = \langle \mathit{send}, m, \mathtt{id} \rangle, \triangleright_d = \langle \mathit{deliver}, m, \mathtt{id} \rangle\}$, where $m \in \mathcal{M}$ and $\mathtt{id}$ is an identifier from an arbitrary totally ordered set that is used to define restrictions on the set of admissible executions of a link (below). Our modelling of links assumes them to be "stateless", thus $\mathcal{S}_{\langle p,q \rangle} = \mathcal{I}_{\langle p,q \rangle} = \{l\}$, with some constant l, and for each message-identifier pair both actions are applicable to l. In the following we denote an execution of the link $\langle p, q \rangle \in \Lambda$ by $\varepsilon|_{p,q}$.

2.3. Plugging together Processes and Links

In this section we discuss how a distributed computation is built from the independent computations of processes and links defined above. In principle, this is achieved by defining an execution ε as an interleaving of some executions of the components (i.e., some $\varepsilon|_p$ and $\varepsilon|_{p,q}$ for all $p \in \Pi$ and $\langle p, q \rangle \in \Lambda$). Clearly, there are a lot of executions which can be constructed in this manner, but which we would not consider valid executions of a distributed algorithm. In order to filter out these unwanted executions we state admissibility conditions.

We define the global state (also called configuration) of a distributed computation as a collection of the states of the components of the system: $\mathcal{S} = \mathcal{S}_{p_1} \times \mathcal{S}_{p_2} \times \cdots \times \mathcal{S}_{p_n} \times \{l\}^{|\Lambda|}$. That is the states of all processes $p_1, p_2, \ldots, p_n$ and the (constant) states of all links. For each p, we denote p's state in some configuration c by $\mathit{state}_p(c)$. Similarly, the set of initial global states $\mathcal{I}$, is built from the set of initial states of processes and links.

The next (global) state relation $\mathcal{N}$ is defined as the "union" of all individual next state relations. Where "union" means that for each $p \in \Pi$ and for any $\langle s, \triangleright, s' \rangle \in \mathcal{N}_p$ there is a $\langle c, \triangleright, c' \rangle \in \mathcal{N}$, with c and c' being two configurations from $\mathcal{S}$, such that the only component that is changed from c to c' is that associated to p. That is, $\mathit{state}_p(c) = s$ and $\mathit{state}_p(c') = s'$ and for all $q \neq p$, $\mathit{state}_q(c) = \mathit{state}_q(c')$. Moreover, the "union" also comprises all state transitions of links.

Based on the above definitions we define executions as alternating sequences of global states and actions. That is, an execution $\varepsilon := c^0, \triangleright^0, c^1, \triangleright^1, \ldots$, where $c^0 \in \mathcal{I}$ and each $\triangleright^i$ is applicable to the configuration c^i. Not all these executions are admissible, however. In order to state more admissibility conditions, we need to introduce the notions of time and faultiness (resp. correctness).

Executions are assumed to be infinite, but we also consider finite prefixes of executions for which we assume that they end in a state. An extension β to an execution prefix α, is an alternating sequence of actions and states such that the concatenation $\alpha\beta$ is a valid execution. Note, that the extension can be finite or infinite leading to $\alpha\beta$ being another execution prefix or an execution.

We assume the existence of Newtonian global time $\mathbb{T} \subseteq \mathbb{R}$.[3] Points in time are denoted by lower case Greek letters. We associate with each action in an execution a point in time, denoted $\chi(\triangleright)$.[4]

Many models assume that some property holds after some amount of time (that is, it holds eventually).[5] In order to ensure that the execution actually reaches this time, we have to preclude what Lynch [1996] calls Zeno behaviour. To this end we require for any admissible execution ε (recall that executions are infinite) that $lim_{i\to\infty}\chi(\triangleright^i) = \infty$. Zeno behaviour occurs when the times of successive events approach but never reach (or pass) a fixed limit point, as in Zeno's paradox of Achilles [Aristotle Physics, 239b 14ff]. Instead of adding this condition one could also require a minimum time between steps (which implies the condition), which is done (implicitly) in models using a discrete time base.[6]

For any point in time τ, we denote with the right continuous function $\mathcal{C}(\tau) \subseteq \Pi$ the set of processes considered correct at τ. For convenience $\mathcal{F}(\tau) = \Pi \setminus \mathcal{C}(\tau)$ denotes the processes considered faulty. The notion of correct and faulty can be naturally generalized for intervals, that is, for any interval $\mathcal{I} = [\tau_1, \tau_2) \subset \mathbb{T}$, we get $\mathcal{C}(\mathcal{I}) = \bigcap_{\tau\in\mathcal{I}} \mathcal{C}(\tau)$ and $\mathcal{F}(\mathcal{I}) = \Pi \setminus \mathcal{C}(\mathcal{I})$. That is, we say a process p is correct in an interval $\mathcal{I}$ if it is not faulty at any time within the interval $\mathcal{I}$; otherwise p is faulty in $\mathcal{I}$.

Note that the functions $\mathcal{C}(\tau)$ and $\mathcal{F}(\tau)$ do not determine the correct and faulty processes, they are just a mean of observation (unlike the failure patterns of Chandra and Toueg [1996]). That is, the function values are dictated by the execution, and not vice versa. Below we define what kind of behaviour causes a process to be faulty at some point in time. When at some point in time τ the behaviour of a process (and its outgoing links) does not abide to what we define below as correct behaviour, the process will not be in $\mathcal{C}(\tau)$. At times where there are no actions the value of the functions cannot change. The sole point of introducing these functions is to simplify the presentation.

Observe that our modelling, like that of the *HO*-model (cf. [Charron-Bost and Schiper 2009]), assumes that faulty processes are always internally correct, and that it is only the transmissions (i.e., the $send$ actions on the link, see below) which are faulty. As mentioned in the introduction, process failures can either be assumed to be permanent, then $\forall\tau, \tau'\, \tau > \tau' \Rightarrow \mathcal{F}(\tau) \supset \mathcal{F}(\tau')$, or intermittent. The fact that the process p can lose its state while being faulty, is captured by adding for each state $state_p$ a special transition $\langle state_p, \langle compute\rangle, recoverystate_p\rangle$ to $\mathcal{N}_p$, with *recoverystate* being a state in the set of recovery states. Whether this set is a singleton or a set containing more than one value, de-

[3] That is, contrary to Dolev et al. [1994] we ignore the effects that the theory of relativity [Einstein 1905] has on distributed computations.

[4] Where χ is the abbreviation of chronos ($\chi\rho o\nu o\varsigma$) the Greek god of time.

[5] In LTL or CTL (cf. eg. [Huth and Ryan 2004]) this is denoted by $\mathsf{F}\mathcal{P}$ or $\diamond\mathcal{P}$, for some predicate $\mathcal{P}$. It has to be noted, however, that in distributed computing literature the term eventual is often used for $\mathsf{FG}\mathcal{P}$ (resp. $\diamond\Box\mathcal{P}$), that is properties that hold eventually forever. Contrasting this, the point of this thesis is to explore properties that hold only intermittently, but at infinitely many times (this is since we do not assume that executions start at some specific time, say $\tau = 0$), leading to $\mathsf{GF}\mathcal{P}$ (resp. $\Box \diamond \mathcal{P}$) style properties.

[6] Unfortunately, this is seldom made clear and the assumption of discrete time is often made under the disguise of "simplicity", even in literature on models with eventual properties, e.g., [Chandra et al. 1996].

pends on the assumptions of the model; in particular, on the assumption on the presence or absence of stable storage (see, e.g., [Aguilera et al. 2000; Hutle and Schiper 2007b]). No admissible execution, is allowed to contain a state transition to *recoverystate* while the process is faulty. When considering an admissible execution ε with intermittent failures, we assume for each finite maximal interval $\mathcal{I} = [\tau, \tau')$ in which p is never correct, there is an action $\triangleright^i \in \varepsilon$ such that $state_p^{i+1} = recoverystate$ and $\nexists \triangleright^j \in \varepsilon|_p : \chi(\triangleright^j) \in \left[\chi(\triangleright^j), \tau'\right)$. That is, we assume that p is in *recoverystate* when it recovers.

Before we now explain the relations between actions on processes and links, we introduce some notation. When an action $\triangleright$ occurs on process p, then we denote this fact by $proc(\triangleright) = p$. Analogously, we define $link(\triangleright) = \langle p, q\rangle$ to denote the link on which $\triangleright$ occurs. In the previous sections we have denoted by $\varepsilon|_p$ (resp. $\varepsilon|_{p,q}$) the execution of a process (resp. a link) by itself. In what follows we extend the meaning of this notation, and denote for any execution ε by $\varepsilon|_p$ the partial execution of p's state machine and by $\varepsilon|_{p,q}$ the partial execution of the state machine of $\langle p, q\rangle$. To obtain a partial execution $\varepsilon|_p$ of some execution ε, we apply the sub-sequence of actions of ε where $proc(\triangleright) = p$ to the initial state of p in ε. Note that the sequence of actions obtained in this way is applicable per definition. Partial executions of links are defined analogously. In literature a partial execution of a process p is also referred to as p's view of the execution.

Moreover, we use these definitions to go into detail regarding the actions occurring at links. Above we have stated that the identifier id used in the actions of links are taken from an ordered set. In the following we let this set be $\mathbb{T}$ and fix the id component of send actions to the time when the action occurs. That is, for any link $\langle p, q\rangle$ and any of its partial executions $\varepsilon|_{p,q}$, we have $\forall \triangleright^i \in \varepsilon|_{p,q} : \triangleright^i = \langle send, m, id\rangle \Rightarrow id = \chi(\triangleright^i)$.

We now explain how such a send event (occurring at a link) is related to the send events of the process at its sending endpoint (i.e., process p for all links $\langle p, q\rangle$). Whenever p is correct there is a one-to-one correspondence between its send actions and the send actions occurring on the outgoing link to the process the message is sent to. In more detail, in any admissible execution $\varepsilon = c^0, \triangleright^0, c^1, \triangleright^1, \ldots$, it must hold that (1) for any sending action $\triangleright^i = \langle send, m, q\rangle \in \varepsilon$ occurring at a process, when $proc(\triangleright^i) \in \mathcal{C}(\chi(\triangleright^i))$, there must be a corresponding sending action $\triangleright^j = \langle send, m, \chi(\triangleright^i)\rangle \in \varepsilon$ on a link, such that $j > i$, $link(\triangleright) = \langle p, q\rangle$. Moreover, (2) for any $\triangleright^j = \langle send, m, \tau\rangle \in \varepsilon$ occurring on the link $\langle p, q\rangle = link(\triangleright^j)$, there must be an action $\triangleright^i = \langle send, m, q\rangle$ with $i < j$ and $\chi(\triangleright^i) = \tau$.

Conversely, when a process is faulty at some point in time, the allowed behaviour depends on its failure mode. In this thesis we consider two possibilities: Byzantine failures and crash failures. For Byzantine failures, there is no restriction. In contrast, all classes of benign failures (to which crash failures belong), the condition (2) has to hold even when a process is faulty. That is, $\forall \langle send, m, \tau\rangle \in \varepsilon|_{p,q}\, \exists \triangleright = \langle send, q, m\rangle \in \varepsilon|_p : \chi(\triangleright) = \tau$. Returning to crash failures, when a process is correct in some interval $[\alpha, \tau)$ and faulty in some interval $\mathcal{I} = [\tau, \beta)$, then we say it crashes at τ, and no messages are sent after τ, i.e., $\forall \tau' \in \mathcal{I}, \tau' > \tau : \nexists \langle send, \_\_, \tau\rangle \in \varepsilon|_{p,q}$.[7]

Before we go on to specify different behaviours of the link, we now consider the relation of deliver actions on processes and their incoming links. Here, we assume that there is always an one-to-one correspondence as for sending actions when a process is correct. That is, in any admissible execution it holds that $\forall \triangleright^i = \langle recv, m, p\rangle \in \varepsilon|_q : \exists \triangleright^j = \langle recv, m, \_\_\rangle \in \varepsilon|_{q,p} : (i > j) \wedge (\chi(\triangleright^i) = \triangleright^j)$ and $\forall \triangleright^j = \langle recv, m, \_\_\rangle \in \varepsilon|_{q,p} : \exists \triangleright^i =$

[7] What happens with messages sent at τ is defined in the next section.

$\langle \mathit{recv}, m, p\rangle \in \varepsilon|_q : (i > j) \wedge (\chi(\triangleright^i) = \triangleright^j)$.

At this point we turn to the meaning of the identifier associated with the deliver events of the links. These are used to discriminate between different behaviours of the link since the messages in $\mathcal{M}$ are not (necessarily) unique. As for the processes, we differentiate between benign and Byzantine behaviour of faulty links. Furthermore, since our definitions should allow links to be faulty intermittently our definitions are based on intervals.

When there is a sending action $\langle \mathit{send}, m, \sigma\rangle \in \varepsilon|_{p,q}$ on the link $\langle p, q\rangle$ and there is exactly one delivery action $\triangleright = \langle \mathit{deliver}, m, \sigma\rangle \in \varepsilon|_{p,q}$ and $\chi(\triangleright) \geq \sigma$, then we say that the message is correctly delivered. In addition to this, if $\chi(\triangleright) - \sigma \leq \delta$ as well, then we say that it was delivered δ-timely. Further, for some interval $\mathcal{I} = [\tau, \tau')$ (with $|\mathcal{I}| \geq \delta$) we say that the link is δ-timely in $\mathcal{I}$, when all messages sent in $[\tau, \tau' - \delta)$ are delivered δ-timely, and no messages are wrongfully delivered by an action $\triangleright$ with in $\chi(\triangleright) \in \mathcal{I}$. A message m is delivered wrongfully, if it is delivered without there being a matching send action; that is, when $\exists\langle \mathit{deliver}, m, \sigma\rangle \in \varepsilon|_{p,q} : \nexists\langle \mathit{send}, m, \sigma\rangle \in \varepsilon|_{p,q}$. Clearly, when no messages are wrongfully delivered the link's behaviour can be called benign. When every message sent at or after some time τ is delivered correctly by a benign link, then we say that a link is reliable from τ on. When τ is at or before the minimal time when a message is sent over a benign link, we say the link is reliable. When τ exists but is not known[8] then the link is said to be eventually reliable.

2.4. Steps

In order to map models used in literature to our generic model, we have to introduce the concept of steps, on which most models are based. The notion of a step is used to define what a process can do at one point in time. (Steps are usually assumed to take zero time.) As we have discussed in Section 1.3, what a process can do in one step depends on the system assumptions. (Clearly, these assumptions imply the number of messages sent or received in one step.) Moreover, computation (and its complexity) is usually assumed to have no influence. Further note that in time-driven models[9] delivery of messages (that is our delivery actions) may occur at any time, it is only send and receive events that define steps.

To simplify the (formal) definition of a step, we first introduce the set $\mathcal{A}_p^{\mathrm{step}} \subseteq \mathcal{A}$ of actions that can only occur in a step (the times when the other actions occur do then no longer matter, and could also be assumed to occur in the some step). Typically, this is the set of all send and reception steps. Next, we first introduce a "restricted" timebase (for an execution ε) $\mathbb{T}_p \subset \mathbb{T}$, which is defined as the set of points in time where some actions occurs at process p. That is, $\mathbb{T}_p := \left\{\tau \in \mathbb{T} \mid \exists \triangleright \in \varepsilon|_p : (\chi(\triangleright) = \tau) \wedge (\triangleright =\in \mathcal{A}_p^{\mathrm{step}})\right\}$. This notion can be generalized to a set of processes $\mathcal{S} \subseteq \Pi$ by defining $\mathbb{T}_{\mathcal{S}} := \bigcup_{p\in\mathcal{S}} \mathbb{T}_p$. Thus,

[8] As usual, the term "known" refers to the fact that there is one value which is the same across all executions. In contrast, a value is not known or "unknown" when for each execution there is a value (but not the same for all). In this specific context, if τ was known, one could simply delay the execution of the algorithm for τ time, thereby guaranteeing the delivery of every message, and thus making the link reliable.

[9] In time-driven models steps are assumed to occur because time passes, while the alternative message-driven models (e.g., [Dolev et al. 1997; Hutle and Widder 2005; Biely and Widder 2009]) assume that steps are triggered by the arrival (i.e., delivery) of a message. This could be expressed by the requirement that every step must start with a delivery action followed by a reception action returning the same message.

$\mathbb{T}_\Pi$ is the set of points in time at which some process performs a send or receive action (that is it takes a step). Then we can define the step that p takes at some time $\tau \in \mathbb{T}_p$ as the collection of actions that occur at τ

$$step_p(\tau) := \{\triangleright \in \varepsilon|_p \,:\, \chi(\triangleright) = \tau\}\,.$$

In order to determine the number of receive respectively send actions that occur in a step of an execution ε we introduce the following abbreviations:

$$\begin{aligned} a_r(p,\tau) &:= \left|\left\{\triangleright^j \in \varepsilon|_p : (\triangleright^j = \langle recv, \_\_, \_\_\rangle) \wedge (\triangleright^j \in step_p(\tau))\right\}\right|, \text{ resp.,} \\ a_s(p,\tau) &:= \left|\left\{\triangleright^j \in \varepsilon|_p : (\triangleright^j = \langle send, \_\_, \_\_\rangle) \wedge (\triangleright^j \in step_p(\tau))\right\}\right|, \end{aligned}$$

As an example, consider the classic partially synchronous model [Dwork et al. 1988]. The steps of which can be described by:

$$\forall p \in \Pi\, \forall \tau \in \mathbb{T}_\Pi : ((a_r(p,\tau) \cdot a_s(p,\tau) = 0) \wedge (a_r(p,\tau) + a_s(p,\tau) \geq 1) \wedge (a_s(p,\tau) \leq 1))\,.$$

Besides bounding the number of certain actions within a step, some models also have assumptions on the order in which the actions in a step are performed. For instance a model might specify that all receiving actions are performed before any sending actions occur. This fact can be specified by

$$\forall \triangleright^i, \triangleright^j : \Big((\triangleright^i = \langle recv, \_\_, \_\_\rangle)) \wedge (\triangleright^j = \langle send, \_\_, \_\_\rangle) \wedge (step(\triangleright^i) = step(\triangleright^j)) \Rightarrow (i < j)\Big)$$

One important aspect of steps we have not touched until now, is that of atomicity. As in database theory (e.g., [Bernstein et al. 1987]) atomicity in distributed computing models refers to an all-or-nothing semantic. That is, either all actions of a step are performed or none. In particular in message-passing systems, the important point is that when a step in which more than one message is sent, then either all messages are sent or none.

As our processes are always[10] internally correct, messages of a step being sent in a step clearly refers to the existence of those send actions on the link that correspond to the send actions performed in the step. For this reason (and to simplify the discussion) we say in the following that a send (not a send action!) is performed when there is a send action on both the process and the link.

Recall that in our definition of crashes above we have stated that when process p crashes at time τ then there are no send actions at any of p's outgoing links *after* τ, while for all times before τ, there is a one-to-one correspondence between send actions of the process and the link.

When one assumes that all sends of a step are performed atomically, then it is sufficient to change that definition to saying that there are no send actions *at or after* τ. Then either all or no messages of a step are sent.[11] In contrast, when steps are not assumed to be atomic and a process crashes at time τ, it is possible that the process crashes at some state $state_p^i$ in the execution, i.e., between two actions, and thus, all sends corresponding to actions $\triangleright^j$ with $j < i$ have a corresponding send action on the link, while the sends, for which send actions $\triangleright^j$ with $j \geq i$ are executed, are not performed.

[10] Or rather nearly always, as the steps going to a *recoverystate* may not be considered correct behaviour.

[11] When all sends are performed at τ, then the process is simply not faulty at τ.

Finally, we note that we assume that the part of the state-machines defined by an algorithm is deterministic. That is, except for delivery and reception events, there is only at most one applicable transition for each process state. The reason for the exemption of delivery events is that delivery events for different messages have to be applicable (otherwise there would only be one message that can be received). Moreover, when more than one message is waiting for reception in the in-buffers of a process, any of them may be received by the next reception event. This leads to the notion of fairness, which (in this case) requires that every message delivered to a process is eventually delivered (unless discarded by a recovery transition).

2.5. Consensus

Although it is not part of the model, we formally define the consensus problem here, as it is central to the remainder of this thesis. In Section 1.4 we have informally discussed the three properties that constitute the consensus problem. We now give the following more formal definition:

Agreement: No two different processes decide differently.

Validity: If all processes initially have the same input value v, then v is the only possible decision value.

Termination: Every correct process eventually decides.

Note that these are — in fact — the properties of uniform consensus. At first sight it seems that solving non-uniform consensus can be defended by arguing that in the context of Byzantine processes faulty processes have to be exempt from agreement. However, as our central assumption about faulty processes is that all processes (nearly) always follow the algorithm, even Byzantine processes should be enabled to reach consensus with the non-faulty processes. Further note that this is only our modelling. When the algorithms are implemented in actual systems (i.e., in practice) a faulty process will (most likely) depart from the assumption that it behaves correctly internally and is therefore also not be able to comply with the three properties above. It would only be required to do so, when it behaves correctly internally.

As discussed above, we model decisions through $\langle \mathit{write}, \mathrm{d}, v \rangle$ actions where the decision value v is irrevocably assigned to d. Moreover, we denote the input value of p, which is part of p's initial state, by $v_{\mathrm{p}}^{\mathrm{I}}$.[12] Based on these notations, we can then reformulate the conditions above more formally:

Termination: $\forall \mathrm{p} \in \Pi : (\nexists \tau \in \mathbb{T} : \mathrm{p} \in \mathcal{F}(\tau)) \Rightarrow (\exists \triangleright^{\mathrm{k}} \in \varepsilon|_{\mathrm{p}} : \triangleright^{\mathrm{k}} = \langle \mathit{write}, \mathrm{d}, \_\_ \rangle)$,

Agreement: $\forall \triangleright^{\mathrm{k}}, \triangleright^{\mathrm{j}} : ((\triangleright^{\mathrm{k}} = \langle d, \mathrm{d}, v \rangle) \wedge (\triangleright^{\mathrm{j}} = \langle d, \mathrm{d}, v' \rangle)) \Rightarrow v = v'$,

Validity: $(\forall \mathrm{p}, \mathrm{q} \in \Pi : v = v_{\mathrm{p}}^{\mathrm{I}} = v_{\mathrm{q}}^{\mathrm{I}}) \Rightarrow (\nexists \langle d, \mathrm{d}, v' \rangle \in \varepsilon : v' \neq v)$, and

Decide Once: $\forall \mathrm{p} \in \Pi : \left| \left\{ \triangleright^{\mathrm{j}} \in \varepsilon|_{\mathrm{p}} \right\} : \triangleright^{\mathrm{j}} = \langle d, \mathrm{d}, \_\_ \rangle \right| \leq 1$.

An alternative validity condition considered in systems with benign failures only, just requires the decision value to be the initial value of some process, or formally:

[12] Alternatively, one could also assume that the input value can be read by a $\langle \mathit{read}, \mathtt{initial}, v \rangle$ action.

Validity': $(\exists \langle d, \mathrm{d}, v\rangle \in \varepsilon) \Rightarrow (\exists \mathrm{p} \in \Pi : v^{\mathrm{I}}_{\mathrm{p}} = v)$.

For arguing about (the impossibility of) consensus algorithms we have to introduce the notions of valency of configurations as well as that of indistinguishability:

Two configurations c and c' are indistinguishable (also called similar) for process p if $\mathit{state}_{\mathrm{p}}(\mathrm{c}) = \mathit{state}_{\mathrm{p}}(\mathrm{c}')$. As usual we denote this fact by $\mathrm{c} \overset{\mathrm{p}}{\sim} \mathrm{c}'$. In distributed computing textbooks (e.g., [Lynch 1996; Attiya and Welch 2004]), indistinguishability is also defined in terms of whole executions or execution prefixes up to some state. That is the two prefixes or executions α and α' are similar to process p when $\alpha|_{\mathrm{p}} = \alpha'|_{\mathrm{p}}$. In case of execution prefixes the two definitions are essentially the same[13] as the state at the end of α may contain the history of the execution up to this point.

Before we discuss valency, we introduce the notion of reachability: A configuration c is reachable from a configuration a when, $\mathrm{c} = \mathrm{a}$ or $\langle \mathrm{c}, \_\_, \mathrm{a}\rangle \in \mathcal{N}$, or there is a configuration b reachable from a such that c is reachable from b. In other words, when c is reachable from a, when there is an extension β to any execution prefix α ending in a, such that $\alpha\beta$ is an execution prefix ending in c. The point in arguing about reachable states in the context of consensus, is that of knowing early (before any processes decide) what the outcome of the distributed computation will be. This knowledge is captured by the notion of valence. If in all configurations reachable from some configuration c the decision value is v, then we call c v-valent, or without referring to the value explicitly, univalent. In impossibility proofs (which are based on binary consensus, that is consensus with only two possible input values) we will also come across bivalent configurations. A configuration c is called bivalent, when processes decide on two different values in the set of configurations reachable from c.

[13] The definition given in the books is stricter (but unnecessarily so).

CHAPTER 3

MODELS AND THEIR RELATIONS

IN THE PREVIOUS chapter we have defined a generic modeling framework for distributed computing models. In this chapter we instantiate our generic model for some partially synchronous models well established in literature. Besides showing how to instantiate well known models in the generic model of Chapter 2, the main aim of this chapter, is to compare the level of abstraction provided by these models, and to determine whether there is a cost associated with these abstractions. To this end we define algorithm transformations, which are based on the idea of expressing the basic operations of one model in terms of the basic operation of another (lower level) model. Here, the basic operations of a model refer to the smallest building block of behaviour a model specifies. Regarding our previous definitions a basic operation lie between actions and steps. Typically an operation corresponds to some actions (of the same type) and some (different) operations form a step. It is possible, however, that an operation represents only a single action or a whole step. As an example, we consider two models: one which allows a broadcast operation to be performed in one step, while the send operation of the other only allows a single message to be sent per step. Then to transform an algorithm for the broadcast model to one for the unicast model we have to replace each broadcast operation by n send operations of the second model, causing step in which the broadcast was performed to be expanded into n steps, when the algorithm is run in the second system.

Apart from the degree of synchrony the most important aspect of the comparison of models is the spatial distribution of synchrony guarantees. Therefore we define families of models, where the members of a family differ in the size of the part of the system where the synchrony assumptions are assumed to hold (eventually). Note that (in contrast to, e.g., [Biely and Widder 2006]) the subset for which the synchrony holds is not known. What is known, is just the minimal size k of the subsets. Thus only the latter could be used in the transformation which we abstain from, however.

The approach taken here is to show that members from different families are equal regarding solvability, when their eventually synchronous subsystem is of the same size. This is achieved by introducing a parameter for this size and then showing that algo-

rithms for a model from one family can be transformed into an algorithm for the model from another family which is parametrized equally. As an example consider two of the models defined below: ROUND_x (a round model) and $\text{ASYNC}+\mathcal{G}_y$ (a failure detector based model). Although they have very different sets of assumptions, we show that as long as $x = y$ they are equivalent regarding solvability. This contrasts similar results for classic synchronous models [Charron-Bost et al. 2000]. In particular, Charron-Bost, Guerraoui, and Schiper have shown that in general, failure detectors do not encapsulate timing assumptions properly: For the perpetually perfect failure detector $\mathcal{P}$ it was shown that the synchronous system has "a higher degree of synchrony" than expressed by the axiomatic properties of $\mathcal{P}$. [1] Being optimistic, one could only hope that (weaker) failure detectors that are just eventually reliable are equivalent to the timing models sufficient for implementing them. Thus the first issue we address in this chapter, is whether being optimistic is justified in the context of eventual synchrony. We answer this questions in the affirmative by providing algorithm transformations among all the families. The second issue is to analyze the cost of such transformations, in order to determine the (relative) "amount of abstraction' in the different models.

3.1. Algorithm Transformations

In this section we introduce the notion of algorithm transformations. In general simulations and transformations are well understood in the context of synchronous as well as asynchronous systems. Roughly speaking, one "system" A simulates another system S, when they produce the same behaviour. In more detail, the set of possible behaviour of the algorithm A that simulates S is a subset of the behaviour possible under the system assumptions in S.

In the case of synchronous systems, the simplifying assumption is made that no additional local computation and no number of additional messages that has to be sent for a simulation will lead to a violation of the lock-step round structure. It follows that layering of algorithms as proposed in [Attiya and Welch 2004, Part II] can be done very easily.

By contrast, for asynchronous systems no time bounds can be violated anyhow. Consequently, the coupling of the algorithm and the underlying simulation can be done so loosely that between any two steps of the algorithm an arbitrary number of simulation steps can be taken. Thus, asynchronous simulations can be very nicely modeled, e.g., via I/O automata [Lynch 1996].

For partially synchronous systems, transformations are not straightforward. Based on the basic operations of a lower model, the operations of higher models must be "implemented" in a more strongly coupled way than in asynchronous systems, while it has to be ensured[2] that the required timing properties are achieved. The important point is what it means to achieve some timing property. When one operation (which is part of an atomic step) is implemented by multiple operations of another model which are not part of the same atomic step, then obviously the atomicity is lost. Achieving the synchrony

[1] This was done by introducing the strongly dependent decision problem, which cannot be solved in asynchronous systems equipped with $\mathcal{P}$, but can be solved in synchronous systems with (explicit) synchrony assumptions. In short, the reason is that when p crashes in an execution possibly after sending one message, $\mathcal{P}$ cannot help q to determine whether p has in fact sent the message or not.

[2] In sharp contrast to the synchronous case, where timing holds per assumption.

assumptions can therefore only mean that the processes (or rather the algorithms) cannot determine the difference between an execution of their transformation and some execution in the original model (this is captured by the concept of traces defined below). Another aspect which is usually a non-issue, but which is of importance for our algorithm transformations is that of well-formedness. Well-formedness captures the fact that an algorithm designer may not always be allowed to perform operations in any order (what that means in detail differs from model to model). For instance, a round based algorithm has to go through the rounds in ascending order, that is, once some operation of round r was executed one is not allowed to perform an operation for some round $r' < r$, or one may also not send round r messages after performing the round r receive operation (cf. Section 3.2.4).

An algorithm transformation $\mathcal{T}_{A \rightharpoonup B}$ generates from any algorithm ALG_A written for some system SYS_A an algorithm ALG_B that for some other model SYS_B by implementing the operations in SYS_A by operations in SYS_B. When we say that a transformation $\mathcal{T}_{A \rightharpoonup B}$ provides an implementation of an SYS_A-operation, this has to be understood as a $\mathcal{T}_{A \rightharpoonup B}$ providing code that replaces that operation,[3] as the new algorithm (the output of the transformation) runs in SYS_B, thus the steps the transformed algorithm takes abide to the synchrony assumptions of SYS_B. For the correctness of such transformations, it has to be shown that ALG_B is well-formed and that the implemented operations are those defined in SYS_A. Moreover it has to be shown that the assumptions on the operations of SYS_A (that ALG_A is based upon) hold for their implementations (given by the transformation) in SYS_B. This is captured by the notion of a trace: The trace of the SYS_A operations is the sequence of implementations of SYS_A operations (in SYS_B) the algorithm calls when being executed via the transformation $\mathcal{T}_{A \rightharpoonup B}$ in SYS_B. The algorithm transformation $\mathcal{T}_{A \rightharpoonup B}$ achieves the assumptions of model SYS_A, when the trace of SYS_A operations is indistinguishable from some execution in SYS_A.

There is one additional condition for an algorithm transformation to be correct: Both the algorithms ALG_A and ALG_B have to solve the same problem. That is, when ALG_A correct with respect to some problem specification in ALG_B has to be correct with respect to the same specification. It is therefore obvious that problem statements only make sense here if they can be stated independently of the model. Consequently, problem specifications cannot contain termination requirements such as "the algorithm terminates after x rounds". This is not model independent as there are (many) models in which there is no (formal) notion of "a round", for example the models of Chandra and Toueg [1996] as well as the two basic models[4] Therefore, we only consider problem specifications which can be specified as tasks [Biran et al. 1990].

Returning to statements like "the algorithm terminates in x rounds", we note that they should be regarded as a property of a particular algorithm and not as part of the problem. Or more to the point, it should be regarded as a statement on the efficiency of the algorithm. Statements such as "the algorithm terminates within an a priori known number of steps" should also be considered performance properties of the algorithm although they seem model independent. Actually this property is what makes an algorithm efficient in the context of models with eventual properties. The actual bound may of course depend

[3] In some sense one can consider an implementation for an operation to provide a macro not a function to be called.

[4] Note carefully that their consensus algorithms are actually analyzed in a round model which is built on the two basic models using clock/round synchronization ([Dwork et al. 1988, Section 5.1]).

on the particular model in use and may change due to a transformation. Even worse, such performance properties may even be lost when the algorithm is transformed. That is, an efficient algorithm is transformed to a non-efficient one.

Obviously, "good transformations" avoid such behaviour, which leads us to the following definition of an efficiency-preserving transformation.

A transformation is efficiency preserving, if there exist values B and D such that the transformation is B-bounded and has D-bounded shift. Where B-bounded and D-bounded shift are defined as follows:

An algorithm transformation being B-bounded, if any step of the higher model SYS_A are implemented by at most B steps of the lower model SYS_B.

The notion of D-bounded shift measures how a transformation behaves with respect to the stabilization time (the point in time from which the eventual properties hold) in the two models involved: Consider some step s from which on SYS_B guarantees its stabilizing properties. This step s is part of the transformation of some step S in SYS_A. A valid transformation has to guarantee that there is a step S' from which on the stabilizing properties of the implemented model (as would be observed by $\mathcal{A}$) are guaranteed to hold. When an execution exists where $S \neq S'$ the transformation is stabilization shifting. Moreover, we say that a transformation has D-bounded shift, when the transformation guarantees that S' does not occur more than D steps in SYS_A after S.

For example, consider an implementation of the eventual perfect failure detector $\diamond\mathcal{P}$ based on some partially synchronous system with global stabilization time τ_{stab}. Since the timing before τ_{stab} is arbitrary, the processes that some process p suspects may be arbitrary at τ_{stab} as well. In particular some non-faulty process q may be suspected by some other process p. Obviously, it takes some time until p notices this error and the set of suspected processes at p becomes accurate. Until this time τ, the algorithm using the failure detector may query the failure detector arbitrarily but finitely often — say x times. That is, in this example the stabilization of the failure detector based model was shifted by x. In general, when $\tau > \tau_{stab}$, the asynchronous model allows p to make an unbounded (but finite) number of steps, i.e., queries to the failure detector. It follows that this transformation has unbounded shift and can therefore not be efficiency preserving (cf. Section 3.2).

Note that, since our definition of efficiency preserving requires known values for B and D which hold in all executions, model parameters which hold in some executions only (e.g., the "unknown" $\underline{\Delta}_?$ and $\underline{\Phi}_?$ in the PARSYNC models defined below) cannot occur in the expressions given for B or D. In contrast, B and D may depend on the size of the system or the minimal size of the eventually synchronous subsystem.

3.2. Models

We base all models discussed in this Chapter on rather conservative choices as the point of this chapter is to explore differences and similarities between different synchrony assumptions and not to find very weak models.

We assume that every process is connected to every other process, that is we assume a fully connected network. All links are assumed to be reliable. Moreover, we consider up to t permanent crash faults. (With permanent faults once a process is faulty it remains faulty forever.) That is:

$$\forall t' > t : \mathcal{F}(\tau') \subseteq \mathcal{F}(\tau).$$

In all the models processes do not have access to global time (or an estimate thereof). The algorithms that provide the transformations between the models must thus base their synchrony guarantees on the inherent synchrony assumptions. These assumptions are only required to hold for (a size k subset of) the correct processes. Here, a correct process means a process which is always correct $p \in \mathcal{C} \Leftrightarrow \forall\tau : p \in \mathcal{C}(\tau)$. For faulty processes we likewise get $p \in \mathcal{F} \Leftrightarrow \exists\tau : p \in \mathcal{F}(\tau)$, with $f = |\mathcal{F}| \leq t$ denoting the number of actually faulty processes in an execution.

The models we consider vary in their level of abstraction. For example, it will turn out that failure detector [Chandra and Toueg 1996] based models can be considered to be at a higher level of abstraction than partially synchronous systems as they abstract away the timing behavior of systems, while round-based models can be seen as being situated at an even higher level as they abstract away how the round structure is enforced.

Moreover, all models below have in common that the synchrony assumptions do not necessarily hold for all processes but only for a subset of those. The models in one model-family only differ in the size of this set, which is denoted by k. Moreover, it should be clear from this, that $1 \leq k$, since for $k = 0$ no synchrony guarantees would hold, and therefore all families would collapse to the asynchronous system with respect to synchrony assumptions (there are still differences in atomicity). Moreover, the synchrony assumptions for the k processes only have to hold eventually, and we require the k processes to be correct. For this reason, k is bounded from the above by $n - f$. Note that f may be different in every execution, as it denotes the actual number of crashes in an execution. In this case, our models coincide with classic models.

3.2.1. The Failure Detector Model

As our first model, we consider an asynchronous model augmented with a failure detector oracle. There are no direct assumptions on the speed of computation or communication. In each step a process, that executes a well-formed algorithms, may execute one, two or all of the following operations in the given order[5]:

***a-receive*$_p$()**: Returns a message m, $\langle m, q\rangle \in \mathcal{M} \times \Pi$, sent from q to p,
***a-query*$_p$()**: Queries the failure detector of p,
***a-send*$_p(m, q)$**: Sends a message m to process q.

The operation *a-receive*$_p()$ allows a process to receive a message/sender tuple, that is p simply performs a $\triangleright_r$ action, without the adversary being restricted in any way beyond the reliable transmissions assumptions. Since send operations are uni-cast in this model, each *a-send*$_p(m, q)$ corresponds to exactly one send action. Finally, the *a-query*$_p()$ maps to an action $\langle \mathit{read}, \mathcal{G}_k, \mathtt{returnvalue}\rangle$, that is a query action for the $\mathcal{G}_k$ failure detector. A failure detector is of class $\mathcal{G}_k$ if it fulfills:

k-Eventual Trust. $\exists\Pi' \exists\tau \forall\tau' > \tau \forall p \in \mathcal{C} : (|\Pi'| \geq k) \wedge (\Pi' \subseteq \mathit{query}_p^{\mathcal{G}_k}(\tau') \subseteq \mathcal{C})$

That is, there exists a set $\Pi' \subseteq \Pi$ consisting of at least k correct processes, such that there exists a time τ from which on the failure detector output of all correct processes is a

[5]Regarding atomicity Chandra et al. [1996] assume steps to be atomic, while Chandra and Toueg [1996] have a more relaxed view. In the context of algorithm transformations this does not make a difference, however.

set of correct processes and a super-set of Π'. The minimal instant τ is called stabilization time.

Regarding the order of events, we get an expression similar to the one on page 20, but somewhat more complicated, since we have to require that querying the failure detector, i.e., reading the special variable $\mathcal{G}_k$ has to occur after possibly receiving messages and before possibly sending a message.

When we discuss this model in the future we will refer to an instantiation of it by ASYNC+$\mathcal{G}_k$, while the family of asynchronous systems augmented with $\mathcal{G}_k$ is denoted by ASYNC+$\mathcal{G}$.

Since this model is based on the asynchronous system, there is no bound on the maximal message (end-to-end) transmission delay except that it is finite. However, this transmission delay includes the time the message is waiting in I, thus implying that the adversary has to be fair in delivering the messages through reception steps/actions.

At first sight our oracle $\mathcal{G}_k$ might seem artificial. As a trust based oracle, it is of course the converse of failure detectors, which output suspicion lists. For example, consider the eventually strong failure detector $\diamond\mathcal{S}$ [Chandra and Toueg 1996], which guarantees that eventually at least one correct process is not suspected by any correct process. The processes that are not suspected by $\diamond\mathcal{S}$ are those that are trusted by $\mathcal{G}_1$ at each process, and the (at least) one process which is not suspected by $\diamond\mathcal{S}$ at any process is the one process in Π' that is in the intersection of the oracle outputs of all correct processes. Moreover, the two other most important classic stabilizing failure detectors are also closely related to $\mathcal{G}_k$, for some k:

$\diamond\mathcal{S}$ [Chandra and Toueg 1996] The eventual strong failure detector is $\Pi \setminus \mathcal{G}_1$.

$\diamond\mathcal{P}$ [Chandra and Toueg 1996] The eventual perfect failure detector is the converse of $\mathcal{G}_{n-f}$, because $\mathcal{G}_{n-f}$ eventually has to output the names of all $n-f$ correct processes, while $\diamond\mathcal{P}$ outputs the names of all f faulty ones.

Ω [Chandra et al. 1996] Finally, the eventual leader election oracle chooses eventually exactly one leader at all correct processes. This corresponds to a (stronger) variant of $\mathcal{G}_1$, where the output always has only one element.

3.2.2. The Partially Synchronous Model

The next two models (discussed in this section and the next) are both based on the partially synchronous models[6] of Dwork et al. [1988]. The central idea in these models is to bound the delay induced by communication and the (relative) speed of processes. These are measured on a global clock, which is defined differently from our real time clock. Although it is inaccessible to the process as well, it "measures time in discrete integer-numbered steps." The simplest way to map that time base onto ours it to require that processes take steps only when the real time clock has an integer value (i.e., we assume that $\mathbb{T}_\Pi \subseteq \mathbb{N}$), and at least one process takes a step whenever the clock shows an integer value. Although this is unnecessarily restrictive, we stick to this definition for the time being. In order to differentiate between our time base and the discrete one, we use underline values for the latter, e.g., $\underline{\tau}$ denotes a point time in the discrete time base.

[6]Note that, we use their models with "fail-stop" faults. Dwork et al. [1988] also consider omission, "authenticated Byzantine", and Byzantine faults, which we do not discuss in this chapter.

In the variants of their partially synchronous model we discuss in this section, Dwork, Lynch, and Stockmeyer assume that the bounds are known[7] but only hold eventually. In contrast the other variant they consider deals with the case where the bounds are unknown but hold always, which, as Chandra and Toueg [1996] point out, is the same (from the algorithmic viewpoint) as requiring that there is an unknown bound, that only holds eventually. Informally speaking the reason for this "equivalence" is that in either case, the bounds have to be learned. This is, typically, done by increasing the values of timeouts as long as they lead to false positives.

Before we discuss how the speed of computation and communication are bounded in detail, we shall take a look at the definition of a step in this model. In each step a process, that runs a well formed algorithm, performs (exactly) one of the following two operations:

par-send$_{\mathbf{p}}(\mathbf{m}, \mathbf{q})$**:** Sends a message m to q.
par-receive$_{\mathbf{p}}()$**:** Returns a set S, s.t. $\emptyset \subseteq S \subseteq \mathcal{M} \times \Pi$ of message-sender tuples.

It is fairly evident that *par-send*$_p(m, q)$ corresponds to exactly one send action, while multiple receive actions are performed for a single *par-receive*$_p()$. To be more exact, we assume that the semantics of the *par-receive*$_p()$ operation is that it returns all messages currently in the receive buffers $I_{\langle \_\_, p\rangle}$. That is p performs receive actions until the receive action returns $\perp$. Which leads us to the following bounds on the numbers of actions:

$$\forall p \in \Pi\, \forall \tau \in \mathbb{T}_p : ((a_r(p,\tau) \cdot a_s(p,\tau) = 0) \wedge (a_r(p,\tau) + a_s(p,\tau) \geq 1) \wedge (a_s(p,\tau) \leq 1))\,;$$

Since there is only one kind of action (besides computation actions which we always allow) in each step, the ordering of actions is a non-issue in this model. Moreover, since the *par-receive*$_p()$ always delivers all pending messages there is no necessity for fairness requirements regarding messages either.

Following [Dwork et al. 1988], the synchrony assumptions of this model are based on the two parameters $\underline{\Delta}_?$ and $\underline{\Phi}_?$ with the following meaning:

We say that $\underline{\Delta}_?$ holds between p and q in the (discrete time) interval $\underline{\mathcal{I}}$ provided that, if a message m is sent to p by q at time $\underline{\tau} \in \underline{\mathcal{I}}$, $\underline{\tau} + \underline{\Delta}_? \in \underline{\mathcal{I}}$ and p performs *par-receive*$_q()$ at time $\underline{\tau}' \geq \underline{\tau} + \underline{\Delta}_?$, m is delivered to p by time $\underline{\tau}'$. When $\underline{\Delta}_?$ holds for all outgoing links of a set of processes, then we say that $\underline{\Delta}_?$ holds for that set.

Moreover, we say that $\underline{\Phi}_?$ holds for a set of processes Π' in the (discrete time) interval $\underline{\mathcal{I}}$ provided that, in any sub interval of $\underline{\mathcal{I}}$ containing $\underline{\Phi}_?$ (discrete) ticks every process in Π' takes at least one step.

Note that, in contrast to [Dwork et al. 1988], we assume that $\underline{\Delta}_?$ holds only for all outgoing links of a subset of the processes (and not for all links). Such processes are typically called sources (cf. [Aguilera et al. 2003; Malkhi et al. 2005; Hutle et al. 2009]). Moreover, our definition of $\underline{\Phi}_?$ also holds for a subset of processes instead of all correct processes. Our definitions clearly require that set to be a subset of the correct processes, that is $\Pi' \subseteq \mathcal{C}(\underline{\mathcal{I}})$.

[7] Recall that, a bound is "known" if there exists one value, say b, such that it holds for all executions. In general, when one considers a predicate $P(b, \varepsilon)$, we get $\exists b\, \forall \varepsilon :\ P(b, \varepsilon)$. In contrast we say that a bound holds but its value is "unknown" if it is only guaranteed that there is a value such that $P(b, \varepsilon)$ hold for every execution: $\forall \varepsilon\, \exists b :\ P(b, \varepsilon)$.

The definitions of bounds for relative computing and communication speed allow us to define models parametrized by some integer k, such that, in every execution there is a set of processes Π' of cardinality at least k for which the following two conditions hold:

k-Partial Sources. Some unknown $\underline{\Delta}_?$ holds forever for the set Π'.
k-Partially Synchronous Processes. Some unknown $\underline{\Phi}_?$ holds forever for the set Π' .

We denote the system model where in all executions there is a set Π' of size at least k such that these two conditions hold as PARSYNC$_k$. The family of all these models is denoted by PARSYNC. Note that in contrast to the approach by Dwork et al. [1988] we never consider message loss, which is the reason for the two bounds to hold from the beginning, and not just from some stabilization time onwards.

3.2.3. The Eventually Synchronous Model

The eventually synchronous model is also variant of a model introduced by [Dwork et al. 1988]. Thus, the steps and operations are the same as those in the model of the previous section (with different names). Thus there is no need to go into detail again, instead we only state the operations and properties of the model:

ev-send$_p(m, q)$**:** Sends a message m to q.
ev-receive$_p()$**:** Returns a set S, s.t. $\emptyset \subseteq S \subseteq M \times \Pi$ of message-sender pairs.

As in the partially synchronous case, we assume for each run the existence of a set of processes Π' of cardinality at least k such that the following two conditions hold:

k-Eventual Sources. An a priori known $\underline{\Delta}_\diamond$ eventually holds forever for the set Π'.
k-Eventually Synch. Processes. An a priori known $\underline{\Phi}_\diamond$ eventually holds forever for the set Π'.

We denote the system model where in all executions there is a set Π' of cardinality at least k such that these two conditions hold as $\diamond$SYNC$_k$, and the family of these models as $\diamond$SYNC. We call the time from which on the properties hold stabilization time and denoted it by $\underline{\tau}_{stab}$.

Observe that, the definitions for "$\underline{\Delta}_\diamond$ holds" and "$\underline{\Phi}_\diamond$ holds" are essentially the same as those for "$\underline{\Delta}_?$ holds" and "$\underline{\Phi}_\diamond$ holds", respectively. The only difference is that some $\underline{\Delta}_\diamond$ and $\underline{\Phi}_\diamond$ hold eventually for all runs covered by an eventually synchronous model, instead of possibly different $\underline{\Delta}_?$s and $\underline{\Phi}_?$s holding for each run. Therefore it holds — by definition — for any k, that every execution in $\diamond$SYNC$_k$ is also an execution in PARSYNC$_k$, thus it follows trivially that any problem solvable in PARSYNC$_k$ is solvable in $\diamond$SYNC$_k$ as well. However, it will in general (i.e., when $\underline{\tau}_{stab} \neq 0$) not hold that $\underline{\Delta}_\diamond = \underline{\Delta}_?$ and $\underline{\Phi}_\diamond = \underline{\Phi}_?$, as in PARSYNC (without message-loss) the bounds hold from the beginning of the execution.

3.2.4. The Round Model

As our last model we introduce a round-based system, where computation proceed in rounds $r = 0, 1, 2, \ldots$. For each round r a process, that executes a well-formed algorithm, may execute at most one send and at most one receive operation. Moreover, a step of a process is composed of the optional receive operation for round r and the optional send operation of round $r + 1$. Note, that steps are not assumed to be atomic here, and are

just used to group together those operations that occur at (approximately) the same time. That is, the round switch from r to $r+1$ occurs in the middle of a step, and we say that this step starts round $r+1$. The operations are defined as:

rd-send$_p(r, S)$**:** Sends a set M of messages to a set of processes P, where $M \times P = S \subset \mathcal{M} \times \Pi$.

rd-receive$_p(r)$**:** Returns a set $\mathcal{R} \subseteq \mathcal{M} \times \Pi \times \mathbb{N}$.

Note that the above definition implies that S contains at most one message m_q for every process $q \in \Pi$. Moreover, we note that each tuple $\langle m, q, r' \rangle$ in the set $\mathcal{R}$ of messages received in round r denotes a message m that q sent to p in round $r' \leq r$.

Clearly, each step contains $|\mathcal{R}|$ receive actions, as well as, $|\mathcal{S}| \leq |\Pi|$ send actions. Note that the adversary is not allowed to deliver messages "from the future", i.e., from rounds $r' > r$, but it is allowed to deliver a round r message in later rounds $r' > r$. Such models are called communication open. In contrast, in a communication closed model [Elrad and Francez 1982] messages from earlier rounds (i.e., late messages) are simply discarded. Examples for this approach include models where slower messages cause omissions such as the Heard-Of Model [Charron-Bost and Schiper 2009] and the model used by Santoro and Widmayer [1989], as well as the round model used in round-by-round failure detector approach [Gafni 1998] and the basic round model in [Dwork et al. 1988].[8] The synchronous lock-step round model (e.g., [Lynch 1996]) is obviously a special case as there are no late messages, which could be discarded. It is important to note that in both cases, the set of steps that start some round r provides a consistent cut (e.g., [Attiya and Welch 2004; Mattern 1992]). This implies that the actual mapping of steps to real-time does not matter, as "a consistent cut [...] can be transformed into a vertical line" [Mattern 1992, Theorem 2.4]. The reason why we have to consider communication open rounds is that we cannot afford to discard late messages as the other models do not allow message loss.

The central assumption of our round model is that:

k-Eventual Round Sources. There is a set Π' of k correct processes, and a round r, such that every message that is sent by some $p \in \Pi'$ in some round $r' \geq r$ is received in round r' by all correct processes.

We denote by ROUND$_k$ the model in which k-Eventual Round Sources is guaranteed, and the family of such models by ROUND.

3.3. Solvability

Based on the definitions above, we will now discuss the relations among the models regarding solvability. By providing a transformation that allows *any* algorithm solving a problem in one model to another model, we show that the problem is also solvable in the latter model.

3.3.1. Possibility of ASYNC+$\mathcal{G}_k \rightharpoonup$PARSYNC$_k$

[8] In the case of Byzantine failures Dwork, Lynch, and Stockmeyer implement a communication open round model above this model in order by using the retransmission scheme of [Srikanth and Toueg 1987a] (which is discussed in more detail in Chapter 5).

```
1:  variables
2:      ∀q ∈ Π : send_p[q], initially ⊥
3:      trusted_p ⊆ Π, initially ∅
4:      buffer_p , undelivered_p FIFO queue with elements in M × Π, initially empty
5:      counter_p , threshold_p ∈ ℕ, initially 0 and 1, resp.

6:  operation a-send_p(m, q)
7:      send_p[q] ← m
8:      update()

9:  operation a-receive_p()
10:     update()
11:     remove all ⊥ entries from undelivered_p
12:     if undelivered_p is empty then
13:         return ⊥
14:     else
15:         return first ⟨m, q⟩ from undelivered_p

16: operation a-query_p()
17:     update()
18:     return trusted_p

19: procedure update()
20:     forall q ∈ Π do
21:         par-send(send_p[q], q)
22:         send_p[q] ← ⊥
23:     append all ⟨m, q⟩ ∈ par-receive_p() to buffer_p
24:     incr(counter_p)
25:     if counter_p = threshold_p then
26:         counter_p ← 0
27:         trusted_p ← {q : ⟨_, q⟩ ∈ buffer_p}
28:         append buffer_p to undelivered_p
29:         buffer_p ← ∅
30:         incr(threshold_p)
```

Algorithm 3.1. *Algorithm transformation for* ASYNC+$\mathcal{G}_k$⇀PARSYNC$_k$.

In the ASYNC+$\mathcal{G}$ models there are no guarantees on the relative processor speeds and the time it takes for the network to transmit a message. Therefore, an ASYNC+$\mathcal{G}_k$ algorithm can solely be based on the assumption that the set returned by *a-query*$_p$ eventually meets the requirements of k-Eventual Trust. As the transformation is to be general we cannot make any assumptions about the algorithm except for that it uses the operations of ASYNC+$\mathcal{G}$. However, the algorithm resulting from the transformation to an algorithm for the PARSYNC$_k$ model can rely on the fact that the assumptions of PARSYNC$_k$ hold (for the subset Π') eventually. Thus, we replace (or, rather, our transformation replaces) the three operations of ASYNC+$\mathcal{G}$ by the new code which is given in Algorithm 3.1. This replacement-code can be seen as an implementation of the $\mathcal{G}_k$ failure detector in PARSYNC$_k$, with the important difference that our implementation is much tighter coupled to the algorithm (as it is an algorithm transformation, which replaces the implemented operations) than the usual loose coupling of an algorithm using a failure detector and the failure detector's implementation. Apart from that difference, we use a straightforward heart-beat based approach. The *update*() function provides the center of all the replacements of ASYNC+$\mathcal{G}$ operations. Here, we always send messages to all processes,

with all of the messages being $\perp$, except if *update*() replaces an *a-send*(m, q), then m is sent to q. The empty messages are thrown away in the implementation of *a-receive*. Since links are reliable in any PARSYNC model, we can conclude that:

Lemma 3.1. *The transformation of Algorithm 3.1 ensures, that every message which appears to be sent through a-send eventually appears to be received through a-receive.*

The reason why we say messages "appear" to be sent and received, is that the operations *a-send* and *a-receive* are never executed in the run of the transformed algorithm. It is only the trace of these operations that we can argue about. In order to conclude the correctness of the transformation, we have to show that our failure detector implementation ensures that the set of process Π' in the definition of PARSYNC (cf. Section 3.2.2) is the set of processes eventually returned by *a-query*$_p$.

Lemma 3.2. *In any* PARSYNC$_k$ *execution of an arbitrary algorithm $\mathcal{A}$ designed for* ASYNC+$\mathcal{G}_k$ *and transformed by* ASYNC+$\mathcal{G}_k \rightharpoonup$PARSYNC$_k$ *transformation of Algorithm 3.1, the property* k*-Eventual Trust holds for the trace of the* ASYNC+$\mathcal{G}_k$ *operations.*

Proof. By the definition of PARSYNC$_k$, there is a set Π' of processes, with $|\Pi'| = k$, so that after some time τ_0, all $p \in \Pi'$ take a step every $\underline{\Phi}_?$ time steps, and messages originating from these processes arrive after $\underline{\Delta}_?$ time steps. From the algorithm of the transformation, we see that every ASYNC+$\mathcal{G}$-operation involves a call to *update*() and therefore sending a message to all processes. That is, every process p sends a message to any process q every $n + 1$ PARSYNC-steps. Therefore at any time after τ_0, any process q can take at most $(n+1)\underline{\Phi}_? + \underline{\Delta}_?$ steps between two consecutive messages m and m' from some process $p \in \Pi'$ being deliverable. Since any process will take at most n send-steps before delivering pending messages through *par-receive*, it follows that q must receive m' at most $((n + 1)\underline{\Phi}_? + \underline{\Delta}_?) + n$ steps after receiving m.

Since *threshold*$_p$ is ever increasing, $\exists\tau, \forall\tau' \geq \tau$: *threshold*$_p(\tau') > (n + 1)\underline{\Phi}_? + \underline{\Delta}_? + n$. Therefore, for all $p \in \Pi$ and for all $q \in \Pi'$ some message from q will be in *buffer*$_p$, when executing line 27. Thus, eventually all processes in Π' are trusted by all correct processes.

On the other hand, every crashed process eventually stops sending messages, and thus eventually there will be no more messages from such a process in *buffer* of a correct process. Thus, eventually, no faulty process is trusted. □

Note that the ever increasing *threshold*$_p$ affects the detection time of the failure detector and thus stabilization of the ASYNC+$\mathcal{G}_k$-algorithm is guaranteed only when *threshold*$_p$ is greater than $\underline{\Delta}_? + (n + 1)\underline{\Phi}_?$ (cf. Section 3.4.7). As these values are unknown there is no general bound for the stabilization shift.

Theorem 3.3. *Algorithm 3.1 transforms any algorithm $\mathcal{A}$ for* ASYNC+$\mathcal{G}_x$ *to an algorithm for* PARSYNC$_x$*, and this transformation is not efficiency preserving.*

3.3.2. Possibility of PARSYNC$_k \rightharpoonup \diamond$SYNC$_k$

Since the properties of $\diamond$SYNC$_k$ imply the properties of PARSYNC$_k$, for this (trivial) transformation it suffices to replace *par-send* with *ev-send*, and *par-receive* with *ev-receive*, respectively. Thus the transformation is clearly 1-bounded. Also note that there is no stabilization shift either (since abiding to known bounds implies abiding to unknown ones). We thus have:

Corollary 3.4. *There exists a* 1*-bounded transformation for a* PARSYNC$_k$ *algorithm to a* $\diamond$SYNC$_k$ *algorithm, without stabilization shift.*

3.3.3. Possibility of $\diamond$SYNC$_k \rightharpoonup$ROUND$_k$

The next transformation we consider is one that transforms any algorithm for $\diamond$SYNC$_k$ to an algorithm for the ROUND$_k$ model. While each $\diamond$SYNC$_k$ model can be instantiated with any values for $\underline{\Delta}_\diamond$ and $\underline{\Phi}_\diamond$, we implement a particular instance $\underline{\Delta}_\diamond = 0$ and $\underline{\Phi}_? = 1$. There are two key ideas involved in achieving this: On the one hand, we let the transformation execute one round of a ROUND model in each simulated $\diamond$SYNC step. On the other hand, we recall that models of the PARSYNC and $\diamond$SYNC families are based on a discrete integer-numbered time base. That is, we use the round numbers as time-base. This is possible since, as we have noted before, all steps of the round model which start some round r form a consistent cut, and can be (do to Theorem 2.4 of [Mattern 1992]) transformed by the rubber band transformation such that they occur at the same time. Therefore, whatever the actual real-time mapping of the ROUND steps is, the processes perceive the computation as if all round were executed at the same times. This argument implies our first lemma of this section:

Lemma 3.5. *In any* ROUND$_k$ *execution of an arbitrary algorithm* $\mathcal{A}$ *designed for* $\diamond$SYNC$_k$ *and transformed by the* $\diamond$SYNC$_k \rightharpoonup$ROUND$_k$ *of Algorithm 3.2, the property* k*-Eventually with* $\underline{\Phi}_\diamond = 1$ *hold for the trace of* $\diamond$SYNC *operations.*

In the following, when we say that an $\diamond$SYNC step corresponds to some round r, we mean that the transformation of the step executes round r. Moreover, when some process p sends a message m to q using *ev-send* which corresponds to some round r_s. As every message sent in some round r is guaranteed to be received in some round $r' \geq r$, when q simulates an *ev-send* corresponding to some round $r_r \geq r_s$, q will receive m, which implies that

Lemma 3.6. *Every message an algorithm sends through the transformation of ev-send provided by Algorithm 3.2 will eventually be received.*

Next we show that $\underline{\Delta}_\diamond = 0$.

Lemma 3.7. *Any* ROUND$_k$ *execution of an arbitrary algorithm* $\mathcal{A}$ *designed for* $\diamond$SYNC$_k$ *and transformed by the* $\diamond$SYNC$_k \rightharpoonup$ROUND$_k$ *transformation of Algorithm 3.2 is indistinguishable from an* $\diamond$SYNC$_k$ *execution with* $\underline{\Delta}_\diamond = 0$.

Proof. This can be seen from an argument similar to the one for Lemma 3.6: Eventually, if a message m is sent by a round source via an *ev-send* corresponding to round r at the sender, the receiver will put it into $buffer_p$ (as return value from *rd-receive*) in the same round, and then the message is delivered via the first *ev-receive* operation after that. By the definition of $\underline{\Delta}_\diamond$ we get $\underline{\Delta}_\diamond = 0$. □

As we have explicated above, $\underline{\Phi}_\diamond = 1$ holds perpetually. Moreover, it can be seen from the proof of Lemma 3.7 that when the ROUND-system stabilizes in round r, then for all *ev-send*'s corresponding to round $r' \geq r$, $\underline{\Delta}_\diamond = 0$ holds. Therefore, we get that

Lemma 3.8. *Algorithm 3.2 has no stabilization shift.*

```
1:  variables
2:      r ∈ ℕ, initially 0
3:      buffer_p ⊆ ℕ × ℳ × Π, initially ∅

4:  operation ev-send_p(m, q)
5:      rd-send_p(r, {⟨m, q⟩})
6:      buffer_p ← buffer_p ∪ rd-receive_p(r)
7:      r ← r + 1

8:  operation ev-receive_p()
9:      rd-send_p(r, ∅)
10:     buffer_p ← buffer_p ∪ rd-receive_p(r)
11:     r ← r + 1
12:     del ← {⟨m, q⟩ | ⟨m, q, _⟩ ∈ buffer_p}
13:     buffer_p ← ∅
14:     return del
```

Algorithm 3.2. *Algorithm transformation for* $\diamond$SYNC$_k \rightharpoonup$ ROUND$_k$.

As each round is spread across 2 steps, it is clear that the transformation is 2 bounded, since we map each $\diamond$SYNC step to one round:

$$\textit{ev-send} \mapsto \begin{cases} \textit{rd-send} \\ \textit{rd-receive} \end{cases} \tag{3.1}$$

$$\textit{ev-receive} \mapsto \begin{cases} \textit{rd-send} \\ \textit{rd-receive} \end{cases} \tag{3.2}$$

Theorem 3.9 summarizes the results of the above discussions about Algorithm 3.2.

Theorem 3.9. *Algorithm 3.2 transforms any algorithm $\mathcal{A}$ designed for* $\diamond$SYNC$_x$ *to an algorithm for* ROUND$_x$*, and this transformation is 2-bounded and does not shift stabilization.*

3.3.4. Possibility of ROUND$_k \rightharpoonup$ASYNC+$\mathcal{G}_k$

For implementing a round structure on top of our failure detector we simply wait in each round until we have received messages for the current round from all the trusted processes. From inspecting the code of the algorithm transformation given in Algorithm 3.3, we see that each *rd-send* is implemented by n *a-send* operations (although these do not necessarily get executed in the order given below), while *rd-receive* may require an unbounded number of steps, yielding that this transformation is unbounded:

$$\textit{rd-send} \mapsto \begin{cases} \textit{a-send}(\_, p_1) \\ \vdots \\ \textit{a-send}(\_, p_n) \end{cases} \tag{3.3}$$

$$\textit{rd-receive} \mapsto \begin{cases} \textit{a-receive} \\ \textit{a-query} \\ \textit{a-receive} \\ \textit{a-query} \\ \vdots \end{cases} \tag{3.4}$$

```
1:  variables
2:    undelivered_p ⊆ ℕ × ℳ × Π, initially empty

3:  operation rd-send_p(r, S)
4:    forall ⟨m, q⟩ ∈ S do
5:      a-send_p(⟨r, m⟩, q)
6:    forall q ∉ {q′ | ⟨m′, q′⟩ ∈ S} do
7:      a-send_p(⟨r, ⊥⟩, q)

8:  operation rd-receive_p(r)
9:    repeat
10:     undelivered_p ← undelivered_p ∪ {a-receive_p()}
11:     trusted_p ← a-query_p()
12:     Q ← {q | ⟨r, _, q⟩ ∈ undelivered_p}
13:   until Q ⊇ trusted_p
14:   del ← {⟨r′, m, q⟩ ∈ undelivered_p | r′ ≤ r ∧ m ≠ ⊥}
15:   undelivered_p ← undelivered_p \ del
16:   return del
```

Algorithm 3.3. *Algorithm transformation for* ROUND$_k$⇀ASYNC+$\mathcal{G}_k$.

Lemma 3.10. *In any* ASYNC+$\mathcal{G}_k$ *execution of an arbitrary algorithm* $\mathcal{A}$ *designed for a* ROUND$_k$ *system and transformed by the transformation given in Algorithm 3.3, the property* k*-Eventual Round Sources holds for the trace of the* ROUND *steps.*

Proof. Following the same pattern as above, we show the lemma by proving that eventually at least the messages from the trusted processes are received by all correct processes in the round they were sent in.

We first observe that every correct process sends (a possibly empty, i.e., $\langle r, \bot\rangle$) message to every other process for each round, therefore progress is ensured, since at some point a message must have arrived from every process currently trusted (cf. line 13).

For each q which is eventually trusted forever, there is a round r for which it is trusted by all processes when executing line 14. Obviously from now on (i.e., for rounds $r' \geq r$) a message from q must be contained in the $undelivered_q$ set of every alive process every time a process reaches this line. If the content of the message is not $\bot$ it will also be delivered, thereby ensuring that every correct process does indeed receive p's round r' message (if it sent one). □

Given that in the models of the ASYNC+$\mathcal{G}$ family links are assumed to be reliable, and since all messages delivered through *a-receive*, are delivered in some round — although this round may possibly be many rounds after it was sent — it is evident that the following theorem follows from Lemma 3.10 and Equations (3.3) and (3.4).

Theorem 3.11. *Algorithm 3.3 transforms any algorithm* $\mathcal{A}$ *for* ROUND$_k$ *to an algorithm for* ASYNC+$\mathcal{G}_k$*, and this transformation is not efficiency preserving.*

By providing algorithm transformations for all intermediate steps in the cycle

$$\text{ASYNC+}\mathcal{G}\rightharpoonup\text{PARSYNC}\rightharpoonup\diamond\text{SYNC}\rightharpoonup\text{ROUND}\rightharpoonup\text{ASYNC+}\mathcal{G},$$

we have shown that all the model families are equally powerful regarding solvability.

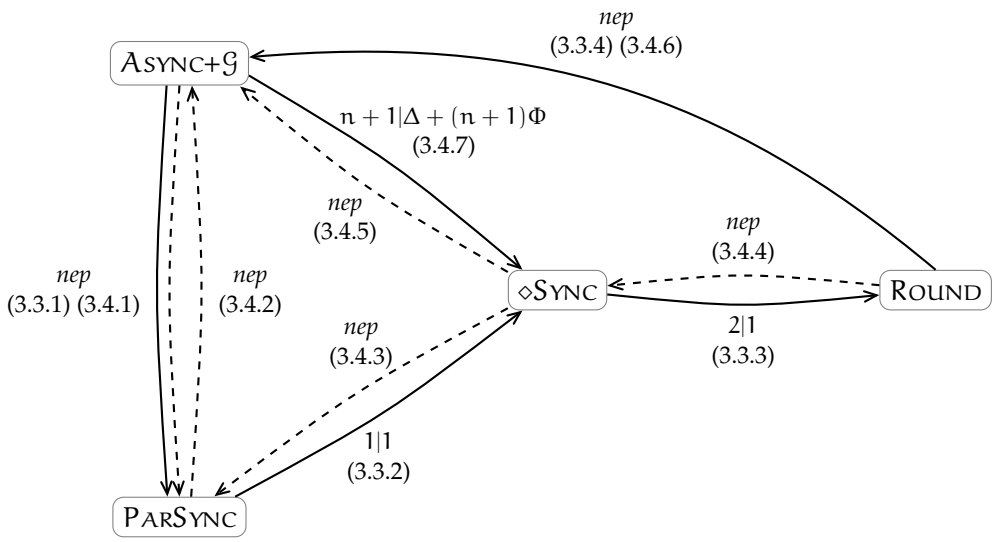


Figure 3.1. *Relations of models with pointers to subsections in this chapter*

3.4. Efficiency of Transformations

In the previous section we showed that from a solvability point of view, all four model families are equivalent. The chain of transformations of a PARSYNC algorithm to a $\diamond$SYNC algorithm to a ROUND algorithm are efficiency-preserving transformations. That means, if there is an efficient algorithm for an PARSYNC model, the transformed algorithm is also efficient in the respective $\diamond$SYNC and ROUND models. In this section we show that there is no transformation that maintains this efficiency for (i) the two other transformations (ROUND$\rightharpoonup$ASYNC+$\mathcal{G}$ and ASYNC+$\mathcal{G}\rightharpoonup$PARSYNC) and (ii) for the opposite directions for the transformations in the chain of efficient transformations. Note, however, that this does not imply the non-existence of efficient algorithms for these models, but just that these cannot be obtained by a general transformation from an efficient algorithm of the other model.

Figure 3.1 gives an overview over the results of the previous section and the efficiency results we will derive in this section. An arrow from model SYS_A to model SYS_B indicates that a result on algorithm transformations from model SYS_A to model SYS_B can be found in the indicated (sub)section. These pointers are given in parenthesis, with two pointers indicating the sections for the possibility result and the impossibility of an efficiency-preserving transformation, respectively. Solid lines indicate upper bounds, dotted lines indicate lower bounds. When a transformation is efficiency preserving, we indicate by $B|D$ that it is B-bounded and has D-bounded shift. Non-efficiency preserving transformations are marked by *nep*.

3.4.1. Lower Bound on ASYNC+$\mathcal{G}_k\rightharpoonup$PARSYNC$_k$

In the cycle of transformations which provided the equivalence of the models regarding solvability we find two transformations which are not efficiency preserving. For one of those, i.e., ASYNC+$\mathcal{G}_k\rightharpoonup$PARSYNC$_k$, we now show that there it is in fact impossible to provide a general transformation for this case. (The other case is shown in Section 3.4.6.)

The main idea behind the proof is that no reliable suspicion of faulty processes can be made within known time in a system with unknown delays.

Theorem 3.12. *There exists no efficiency-preserving transformation that transforms any algorithm $\mathcal{A}$ for* ASYNC+$\mathcal{G}_k$ *to an algorithm for* PARSYNC$_k$.

Proof. We do so by assuming the contrary, i.e., we assume that there is a efficiency-preserving transformation. Using such a transformation (being is bounded and having bounded shift) every process must stop suspecting all sources after a bounded number of steps. Let L be such a bound, and $\underline{\tau}_L$ the earliest time where all correct processes made L steps. By this time, every correct process has to have no faulty process in its trusted list, and at least k processes have to be the same for all processes. Consider now a run R, where no message from a correct processes arrives before $\underline{\tau}_L$, that is a run where $\underline{\Delta}_? > \underline{\tau}_L$. Let Q be the set of processes that are trusted by some correct process at τ_L in R. Clearly, we have $|Q| \geq k$, as $\Pi' \subseteq Q$, with $|\Pi'| \geq k$ as required by the definition of ASYNC+$\mathcal{G}_k$. Now pick one of these processes, say q, and consider a run R' that is similar to R up to some $\underline{\tau} > \underline{\tau}_L$, except that q has crashed initially. Since R' is indistinguishable from R up to $\underline{\tau}_L$ (no messages arrive from q in R and R' until $\underline{\tau}_L$), q will be trusted by all processes, which contradicts the assumption that only correct processes are trusted at this time. Thus, $\underline{\tau}_L$ cannot be bounded, and consequently the transformation cannot be efficiency preserving. □

3.4.2. Lower Bound on PARSYNC$_k$→ASYNC+$\mathcal{G}_k$

Next, we show that the transformation in the opposite direction (with respect to the transformation considered in the previous subsection) cannot be efficiency preservingeither. Here, the main idea is that no message with an unbounded delay can be received in a bounded number of steps.

Theorem 3.13. *There exists no efficiency-preserving transformation that transforms any algorithm $\mathcal{A}$ for* PARSYNC$_k$ *to an algorithm for* ASYNC+$\mathcal{G}_k$.

Proof. Assume by contradiction that there exists a transformation $\mathcal{T}$ that is B-bounded for some B and has bounded shift. By the definition of PARSYNC, there exists a $\underline{\Delta}_?$ and a set of processes Q, with $|Q| = k$, such that all messages sent by a $q \in Q$ are received in the first reception step $\underline{\Delta}_?$ after being sent at any correct p. Assuming that q's sending action for some message m sent at PARSYNC time $\underline{\tau}$ happens at real time τ (i.e., the timebase of ASYNC+$\mathcal{G}_k$), the message m is received in the simulated *par-receive* step at $\underline{\tau} + \underline{\Delta}_?$ or the first such step thereafter. Let the real time when this step occurs be denoted by τ', which has to be before p has performed $B\underline{\Delta}_?$ ASYNC+$\mathcal{G}_k$ steps after q sent the message τ. This, however, contradicts the assumptions of ASYNC+$\mathcal{G}_k$, which states that there is no guarantee that a message is received within any bounded time after being sent. □

3.4.3. Lower Bound on $\diamond$SYNC$_k$→PARSYNC$_k$

While the transformation of PARSYNC$_k$ algorithms to ones for the $\diamond$SYNC$_k$ model is rather simple (recall that by definition every $\diamond$SYNC$_k$ execution is also a PARSYNC$_k$ execution) there is no efficiency-preserving transformation for the opposite direction. The reason for this is, informally speaking, that fixed bounds ($\underline{\Delta}_\diamond$ and $\underline{\Phi}_\diamond$) have to be ensured in a system where there are only unknown bounds ($\underline{\Delta}_?$ and $\underline{\Phi}_?$).

Theorem 3.14. *There exists no efficiency-preserving transformation that transforms any algorithm $\mathcal{A}$ for $\diamond\text{SYNC}_k$ to an algorithm for PARSYNC_k.*

Proof. Assume by contradiction that there exists a transformation $\mathcal{T}$ that is B-bounded for some known B. By the definition of $\diamond\text{SYNC}_k$, there exists a known $\underline{\Delta}_\diamond$, a time $\underline{\tau}^0_\diamond$, and a set of processes Q, with $|Q| \geq k$, such that all messages sent by some $q \in Q$ at $\underline{\tau}_\diamond \geq \underline{\tau}^0_\diamond$ are received by the first reception step following $\diamond\text{SYNC}$ time $\underline{\tau}_\diamond + \underline{\Delta}_\diamond$ at any correct process p. Assume that the sending of such a message m at $\diamond\text{SYNC}$ time $\underline{\tau}_\diamond$ is implemented by a *par-send* operation executed at PARSYNC time $\underline{\tau}_?$. By the $\diamond\text{SYNC}_k$ assumptions, the message has to be delivered in the first *ev-receive* step happening at or after $\underline{\tau}'_\diamond = \underline{\tau}_\diamond + \underline{\Delta}_\diamond$, with the $\diamond\text{SYNC}$ time $\underline{\tau}'_\diamond$ corresponding to PARSYNC time $\underline{\tau}'_?$. On the one hand we have that $\underline{\tau}'_? \leq \underline{\tau}_? + B\underline{\Delta}_\diamond$, because the transformation is B-bounded. On the other we have that $\underline{\tau}'_? \geq \underline{\tau}_? + \underline{\Delta}_?$, in order to guarantee that the message can be delivered. Clearly, this is impossible to achieve in runs where $\underline{\Delta}_? > B\underline{\Delta}_\diamond$, which is a legal choice in the PARSYNC_k model. □

3.4.4. Lower Bound on $\text{ROUND}_k \rightarrowtail \diamond\text{SYNC}_k$

The key to our next lower bound is, that before stabilization time, round numbers can get arbitrarily far apart, and that in order to be efficiency preserving a transformation has to repair this within bounded time (which is impossible).

Theorem 3.15. *There exists no efficiency-preserving transformation that transforms any algorithm $\mathcal{A}$ for ROUND_k to an algorithm for $\diamond\text{SYNC}_k$.*

Proof. Assume by contradiction that a transformation $\mathcal{T}$ exists for $\text{ROUND}_k \rightarrowtail \diamond\text{SYNC}_k$ which is B bounded and has D-bounded shift. Consider what happens before the stabilization time $\underline{\tau}_{stab}$: some correct processes p may drift away from another correct process q by d rounds (that is at $\underline{\tau}_{stab}$ p's round number is greater than q's by d). Clearly d is unbounded, since the transformation is B bounded and $\underline{\Phi}_\diamond$ does not hold yet. In particular, consider the execution where q takes no steps before $\underline{\tau}_{stab}$, but p takes dB steps. Then p has reach at least round d, while p is still in round 0. Now, consider what happens after $\underline{\tau}_{stab}$.

We assume that q is one of the processes in Π' for which the synchrony assumptions eventually hold. In order for q to be one of the k round sources, it is required that there exists some number of rounds D (since the transformation is assumed to be D-bounded) such that after going through D rounds (any) p will receive q's message within the round in which it was sent. It is clearly necessary that $D > d$ and since d is unbounded this is not possible. Therefore, q cannot be one of the k round sources. Thus, when $k = n - f$, we are done.

Otherwise, i.e., $k < n - f$, some s has to be a round source although it is not one of the processes in Π for which the synchrony assumptions eventually hold. Now assume that s is indeed a round source up to or in some round r, that is, we assume that q receives the message s sends in round r in round r. As the transformation is B bounded, q has to switch to round $r+1$ after taking B steps at the latest. But s is not guaranteed to take steps at any particular rate, therefore s may not take any steps until q has switched to round $r + 1$ or, for that matter, any round $r' > r$. Thus, s cannot be a round source perpetually.

Since s was an arbitrary process which is not in the set Π' of the k processes for which some timing is guaranteed, none of these can become round sources through the transformation. Moreover, above we have shown that q is not a round source either. Thus

there could be at most $k-1$ round sources, which clearly contradicts the assumption that $\mathcal{T}$ transforms $\textsc{Round}_k \rightarrowtail \diamond\textsc{Sync}_k$. Thus, no transformation can be efficiency preserving. □

3.4.5. Lower Bound on $\diamond\textsc{Sync}_k \rightarrowtail \textsc{Async}+\mathcal{G}_k$

Our next lower bound follows from two previous bounds: an upper bound and a lower bound. Corollary 3.4 states that there is an efficiency-preserving algorithm transformation for $\textsc{ParSync}_k \rightarrowtail \diamond\textsc{Sync}_k$. If there was an efficiency-preserving transformation for $\diamond\textsc{Sync}_k \rightarrowtail \textsc{Async}+\mathcal{G}_k$, then these two could be combined to obtain a efficiency-preserving transformation for $\textsc{ParSync}_k \rightarrowtail \textsc{Async}+\mathcal{G}_k$, which is impossible since such transformation cannot exist due to Theorem 3.12. Thus, we can deduce our Corollary 3.16:

Corollary 3.16. *There exists no efficiency-preserving transformation that transforms any algorithm $\mathcal{A}$ for $\diamond\textsc{Sync}_k$ to an algorithm for $\textsc{Async}+\mathcal{G}_k$.*

3.4.6. Lower Bound on $\textsc{Round}_k \rightarrowtail \textsc{Async}+\mathcal{G}_k$

Our last lower bound result is a corollary to previous results as well, with the argument following the same line. From the existence of an efficiency-preserving transformation for $\diamond\textsc{Sync}_k \rightarrowtail \textsc{Round}_k$ (Theorem 3.9) and the nonexistence of an efficiency-preserving transformation for $\textsc{Async}+\mathcal{G}_k$ (Corollary 3.16) we can conclude that there is no efficiency-preserving transformation for $\textsc{Round}_k \rightarrowtail \textsc{Async}+\mathcal{G}_k$:

Corollary 3.17. *There exists no efficiency-preserving transformation that transforms any algorithm $\mathcal{A}$ for $\textsc{Round}_k$ to an algorithm for $\textsc{Async}+\mathcal{G}_k$.*

3.4.7. Upper Bound on $\textsc{Async}+\mathcal{G}_k \rightarrowtail \diamond\textsc{Sync}_k$

To show that an efficiency-preserving transformation which transforms algorithms for $\textsc{Async}+\mathcal{G}_k$ to algorithms for $\diamond\textsc{Sync}_k$, we use a modified version of Algorithm 3.1. The only modification necessary is concerned with $\mathit{threshold}_p$: it is initialized to $\underline{\Delta}_\diamond + (n+1)\underline{\Phi}_\diamond$ and the increment of the threshold (i.e., line 30) is omitted. This incorporates the fact that we know $\underline{\Delta}_\diamond$ and $\underline{\Phi}_\diamond$ in advance, and thus there is no need for the algorithm to try to estimate it.

It is clear that the proof of Algorithm 3.1 applies analogously here and thus this transformation is correct. It is also easy to see that it is also $(n+1)$-bounded:

$$\mathit{a\text{-}send}, \mathit{a\text{-}receive}, \mathit{a\text{-}query} \mapsto \begin{cases} \mathit{ev\text{-}send}(\_\_, 1) \\ \vdots \\ \mathit{ev\text{-}send}(\_\_, n) \\ \mathit{ev\text{-}receive} \end{cases} \tag{3.5}$$

It remains to show that the stabilization shift caused by the transformation can be bounded. To see this, note that since every source s will send a message to p in every $\textsc{Async}+\mathcal{G}$ operation (taking at most $(n+1)\underline{\Phi}_\diamond$ $\diamond\textsc{Sync}$ steps) and messages are delivered within $\underline{\Delta}_\diamond$ $\diamond\textsc{Sync}$ steps, p will always receive a message from s before updating $\mathit{trusted}_p$, thus never suspecting s anymore. Therefore the maximal shift is bounded by $\underline{\Delta}_\diamond + (n+1)\underline{\Phi}_\diamond$ $\diamond\textsc{Sync}$ steps, and consequently:

Theorem 3.18. *There exists an efficiency-preserving transformation that transforms any algorithm $\mathcal{A}$ for* ASYNC+$\mathcal{G}_k$ *to an algorithm for* $\diamond$SYNC$_k$ *.*

3.5. Discussion

Solvability. Returning to Figure 3.1 it can easily be seen, the directed sub-graph consisting of solid arrows is strongly connected. As already mentioned above, it follows that all models (with the same size of the eventually synchronous subsystem) are equivalent regarding solvability.

Relation to the results of Charron-Bost, Guerraoui, and Schiper [2000]. Setting $k = n - f$ in our models, the asynchronous model with $\mathcal{G}_{n-f}$ is the asynchronous model augmented with the eventually perfect failure detector [Chandra and Toueg 1996]. Moreover, recall that ROUND$_{n-f}$ is in fact the classic basic round model by Dwork et al. [1988]. We consider the case where $k = n - f$ here, as this is the case where these stabilizing models are equivalent from a solvability viewpoint while it has been shown that their perpetual counterparts [Charron-Bost et al. 2000] are not.

The main observation in the relation to the work of Charron-Bost et al. [2000] is concerned with the term "eventually" in the model definition: In the asynchronous model augmented with $\mathcal{P}$ [Chandra and Toueg 1996], two things happen in every execution: (i) eventually, the failure detector outputs the set of faulty processes, and (ii) eventually, the last process crashes and all its messages are received. However, there is no guarantee about the relation of these two points in time. That is, if a process p crashes, $\mathcal{P}$ does not provide information on the fact whether there are still messages sent by p in transit (cf. [Gärtner and Pleisch 2002]), while perpetual timing information allows to determine this (by using timeouts). When timing guarantees are only eventual, then just as in the case of $\mathcal{P}$, there is no way to reliably determine whether messages are still in transit or not, as the time from which on the guarantees hold may still lie in the future.

Timing Assumptions. Our results show equivalence of diverse models with stabilizing properties, where the properties that are guaranteed to hold have the same spatial distribution. Informally, for the models we present, it is equivalent if a source is defined via timing bounds, restrictions on the rounds in which its messages are guaranteed to be received, or whether the "source" has just the property that it is not suspected by any process. Consequently, we conjecture that similar results hold for other timing assumptions as the FAR model [Fetzer et al. 2005] or models where the function in which timing delays eventually increase is known given that the spatial distribution of timing properties is the same as above.

Number of Timely Links. An interesting consequence of our results concerns models which require only t links of a source to be eventually timely (e.g., [Aguilera et al. 2004]). In models stronger than the asynchronous one where at least one such source exists Ω (and thus $\diamond\mathcal{S}$) can be implemented. As $\diamond\mathcal{S}$ is equivalent to $\mathcal{G}_1$, our results reveal that one can transform any algorithm which works in PARSYNC$_1$ (one source with n timely links) to an algorithm which works in a partially synchronous systems with a source with $t < n$ timely links. Consequently, although the number of timely links was reduced in the model assumptions, the set of solvable problems remained the same. That is, from the viewpoint of solvability the quest [Aguilera et al. 2004; Malkhi et al. 2005; Hutle et al.

2009] for the weakest model for implementing Ω has not led to weaker models. (Which is not too surprising as they all allow implementing the same failure detector.)

Efficiency. An algorithm solving some problem in a stabilizing model is said to be efficient, if it decides (i.e., terminates) after a bounded number of steps after stabilization. Consider some efficient algorithm ALG_A working in model SYS_A. If there exists an efficiency-preserving transformation from SYS_A into some model SYS_B, this implies that there exists an efficient algorithm in SYS_B as well (the resulting algorithm when ALG_A is transformed). The converse, however, is not necessarily true. The (dotted) lines with a *nep* label in Figure 3.1 show that no efficiency-preserving transformations exist. Consequently, by means of transformations (which are general in that they have to transform all algorithms) nothing can be concluded about the existence of an efficient solution in the absence of an efficiency-preserving transformation from SYS_A to SYS_B. As an example consider ROUND_{n-f} and $\diamond\text{SYNC}_{n-f}$ for which efficient consensus algorithms were given in [Dutta et al. 2007] and [Dutta et al. 2005], respectively. Despite this fact, our results show that there cannot be a (general) efficiency-preserving transformation from ROUND_k to $\diamond\text{SYNC}_k$ algorithms.

Temporarily synchronous systems. In the models considered above it is assumed that some assumptions on synchrony hold from some stabilization time onwards forever. As we know this is not necessary for all problems (for instance, for consensus it is sufficient for the system to hold for some time; e.g., [Dwork et al. 1988; Hutle and Schiper 2007b; Charron-Bost and Schiper 2009]). Indeed, as Dwork, Lynch, and Stockmeyer put it, "In realistic situations, the upper bound cannot be reasonably be expected to hold forever after [stabilization time], but perhaps only for a limited time." [Dwork et al. 1988, p290]

In this spirit, they show that their algorithms are guaranteed to decide within polynomially bounded time. Let $L = 4(n+1)$ and GSR denote the stabilization round, i.e., the one from which the $(n-f)$-Eventual Round Sources property holds, then Dwork, Lynch, and Stockmeyer show that all correct processes make their decision by round $\text{GSR} + L$ of their "basic round model". (This round model is similar to our ROUND_{n-f}.) When reliability of the message transmission does not hold for any rounds $r > \text{GSR} + L$, this can clearly not change the outcome of the consensus algorithm. Dwork, Lynch, and Stockmeyer then show that in their eventually synchronous model (which is the same as our $\diamond\text{SYNC}_{n-f}$) there is a polynomially bounded number of $\diamond\text{SYNC}$ steps between stabilization time and the decision of the last correct process. Clearly, this seems to be in contradiction to our Theorem 3.15, which states that no general algorithm transformation for $\text{ROUND}_k \rightarrowtail \diamond\text{SYNC}_k$ which is efficiency preserving exists. The important point to note here, is that in the simulation of the basic round model, when a process switches from round h to round k all rounds $h, \ldots, k-1$ are simulated within one step (by not actually sending any messages in the send steps and not receiving any messages in the receive steps of these intermediate rounds). When the number of rounds "skipped" in this way may be unbounded, the simulation has unbounded shift in our framework, thus there is no contradiction here.

Finally, we turn our attention to the failure detector based models of the $\text{ASYNC}+\mathcal{G}$ family. As for any other asynchronous system augmented with a failure detector it is impossible to define a failure detector which (i) has a property that only holds intermittently but (ii) which the algorithm can exploit. The reason for this is, that while (i) is possible, i.e., it is possible to define some property on the failure detector history $H(p,t)$

to hold within some time interval $\mathcal{I}$ only, (ii) is not, as p cannot be required to take a step during $\mathcal{I}$ without abandoning the asynchronous system. This is reflected by the fact that all transformations to the ASYNC+$\mathcal{G}$ family are not efficiency preserving.

Conversely, consider two models SYS_A and SYS_B where an efficiency-preserving transformation $\text{SYS}_A \rightharpoonup \text{SYS}_B$ exists: here it is clear, that when an algorithm solves some problem within x steps of SYS_A after SYS_A stabilizes then the transformed algorithm is guaranteed to solve the problem within $D + x \cdot B$ SYS_B-steps after SYS_B has stabilized. From this it follows that when we consider a system SYS'_A in which the eventual properties of SYS_A hold for $y > x$ steps only, instead of forever[9], then we can conclude from our efficiency discussions that the same problems solvable in SYS'_A are solvable in SYS'_B as well, with SYS'_B being a variant of SYS_B with the eventual properties holding for $D + x \cdot B$ instead of forever.

[9]Note that, SYS_A cannot be ASYNC+$\mathcal{D}$ for some failure detector D by definition, resp. by the discussions above.

PART II

IMPOSSIBILITY RESULTS AND LOWER BOUNDS

CHAPTER 4

DYNAMIC LINK FAILURES

IN THIS CHAPTER we turn our attention to finding a lower bound on the resilience for consensus in the presence of both link and process failures. Link failures will be allowed to be dynamic and faulty links may either drop or modify messages. Process failures are assumed to be permanent Byzantine failures, i.e., static.

Since unrestricted dynamic link faults make consensus impossible to solve (cf. [Santoro and Widmayer 1989]) we have to restrict them. In contrast to other models [Pinter and Shinahr 1985; Sayeed et al. 1995; Siu et al. 1998b] we do not restrict the total number of link faults in an execution, neither do we take the approach of mapping link failures to (the sending or receiving) processes [Gong et al. 1995; Perry and Toueg 1986].[1] Under such assumptions, transient link failures (hitting different links in different rounds) lead to quickly exhausting the total number of tolerated failures. Such failures call for more flexibility in counting the total number of failures. Similar to the underlying idea of round-by-round failure detectors [Gafni 1998] or the predicates in the HO-Model [Charron-Bost and Schiper 2006a; Charron-Bost and Schiper 2009], we will assume a failure bound on a per round basis.

4.1. Chapter Outline

In order to develop our lower bound (Theorem 4.9), we will start with restating an impossibility results by Fischer et al. [1986] and Schmid and Weiss [2002]. The former is concerned with static Byzantine processes only. The latter is concerned with a single link failure per round/process. Both results are based on same proof techniques. We then generalize this result for larger numbers of link failures (or larger systems) with the same technique. Although this generalization is not completely new, as some of the results were also shown in [Schmid and Weiss 2002] and [Schmid et al. 2009] using other proof techniques, it makes the proofs easier to understand by using "easy impossibility proofs." and also allows to develop a novel unified lower bound for link and process faults. In fact, until now lower bounds were either concerned with links or process failures and it was (implicitly) assumed that for systems where both a possible the two bounds only need to be added. This, however, is not the case as we will show: Considering the fact

[1] For a more detailed discussions on other models for link failures see Chapter 7.

that consensus requires $n > 3t$ when there are up to t Byzantine process failures, we will see (in Chapter 5) that it is possible to mask link failures given that $n > 2t+4\ell$. The lower bound derived in this chapter is therefore tight.

4.2. Model

Regarding the model we can be very conservative in our choices, as we are interested in a (strong) lower bound. Thus, we assume that processes are either always correct or always faulty, i.e., the set of correct processes never changes: $\forall \tau \in \mathbb{T} : \mathcal{C}(\tau) = \mathcal{C}$. We assume that there is a bound t on the number of Byzantine faulty processes. Moreover, we consider computations of broadcast-based algorithms organized in communication-closed rounds in this chapter, that is, we assume that all messages that are not lost arrive within the round they were sent in. Messages may be lost, created, or changed subject to the following conditions which are derived from the modelling of Schmid and Weiss [2002]:

Firstly, when some non-faulty process p broadcasts a message in round r, then at most ℓ^s processes may not receive exactly the message sent by p, with at most $\ell^{sa} \leq \ell^s$ receiving a corrupted message.

Secondly, in every round every process is guaranteed to receive exactly the messages sent by at least $n - \ell^r$ processes. Of those messages where this is not the case at most $\ell^{ra} \leq \ell^r$ may be corrupted.

Clearly the numbers for the sending and receiving side of a link (ℓ^s, ℓ^{sa} and ℓ^r, ℓ^{ra}, respectively) are not independent, as each link failure occurs on a link between a send and a receiver. Thus in each round the total number of link failures counted on the outgoing links of all processes have to be the same as the total number of link failures occurring on the incoming links of processes. For this reason we make the simplifying assumption that $\ell^r = \ell^s$ and $\ell^{ra} = \ell^{sa}$, which we abbreviate by ℓ and ℓ^a respectively. Moreover, note that, while in the case of $n = 2$ and $\ell = 1$ the one of the two conditions could be omitted. In the general case this is not the case, however.

If $\ell > \ell^a$ some links are guaranteed purely omissive when ℓ link failures occur with respect to one process and round. We abbreviate the number by $\ell^o = \ell - \ell^a$.

Note that in these restriction of link failures there is no relation between the sets of faulty links in different rounds. With regard to the processes, the restrictions are of course not fully independent, as each link failure counts towards the bounds on both the in-bound and outbound process. Consider for instance a link connecting p to q. When this link corrupts a message in some round r, then it counts towards the ℓ^a outgoing links of p that may be faulty in r and towards the ℓ^a incoming links of q that may be faulty in r. Moreover, the modeling of link failures above does of course not restrict the messages that faulty processes send/receive.

Throughout this chapter, we will only consider binary consensus, that is the set of possible initial values $\mathcal{V}$ only includes 0 and 1. This not only simplifies the presentation, but also provides the strongest impossibility result, as there is no way to solve consensus with $|\mathcal{V}| > 2$ when binary consensus is impossible. (Otherwise, such a solution could be used to solve the binary problem.) Moreover, we consider the Validity property for non-benign systems, that is, we require that, when all correct processes use the same input value v, then v must be the decision value.

4.3. Byzantine Resilience Lower Bound

The classic result we present in this section is concerned with the number of processes needed to solve consensus in the presence of up to t permanent Byzantine faults, and no link faults.

The first proof for the fact that at least $3t + 1$ processes are necessary was presented by Pease et al. [1980] based on a problem closely related to consensus: Interactive consistency is the problem of consistently determining the vector of all input values —instead of only one value in case of consensus. It can easily be seen that both problems are equally hard to solve: On the one hand, any deterministic function choosing one of n values will allow solving consensus based on interactive consistency. On the other hand, processes can achieve interactive consistency by broadcasting their initial value and then reaching consensus over what each process has broadcast separately.

Pease et al. [1980] base their lower bound on three scenarios. While the first and third lead to different decisions (due to different initial values), they are both indistinguishable to for some processes to a intermediate scenario, which leads to a contradiction as not all results can be the same. Later, Fischer et al. [1986] presented another proof which in its essence proceeds in the same way, but rather than arguing about scenarios they adapted the technique used by Angluin [1980] to show the impossibility of solving leader election in anonymous rings. Following Fischer, Lynch, and Merritt the technique has become known by the term "easy impossibility proofs". It is based on the fact that the behaviour of a process can depend only on what is locally perceptible. This assumption can be captured by the Locality Axiom below [Fischer et al. 1986]:

Locality Axiom: Let $\mathcal{G}$ and $\mathcal{G}'$ be two distributed systems (represented by graphs) and ε and ε' two executions. Moreover, let $\mathcal{U}$ and $\mathcal{U}'$ be two isomorphic subsystems in $\mathcal{G}$ and $\mathcal{G}'$, respectively. If the corresponding behaviours of the in-edge borders of $\mathcal{U}$ and $\mathcal{U}'$ are identical in ε and ε', then the executions $\varepsilon|_{\mathcal{U}}$ and $\varepsilon'|_{\mathcal{U}'}$ are identical as well [Fischer et al. 1986].

In our framework the last part can be expressed by

$$\begin{aligned}&(\forall\langle p, q\rangle \in \Lambda, \forall\langle p', q'\rangle \in \Lambda : \ (p \notin \mathcal{U} \wedge q \in \mathcal{U} \wedge p' \notin \mathcal{U}' \wedge q' \in \mathcal{U}') \wedge \\ &\quad (\forall\tau : I_{\langle p,q\rangle}(\tau) = I_{\langle p',q'\rangle}(\tau))) \\ &\Rightarrow \varepsilon|_{\mathcal{U}} = \varepsilon'|_{\mathcal{U}'}.\end{aligned}$$

This locality property of executions allows to utilize the notion of a covering. A graph $\mathcal{H}$ is a covering of $\mathcal{G}$ if the vertices of $\mathcal{H}$ can be mapped (surjectively) onto the vertices of $\mathcal{G}$, such that the neighbours of any vertex v in $\mathcal{H}$ are mapped to neighbours of v's image in $\mathcal{G}$. That is—as Angluin [1980] puts it—"if $\mathcal{H}$ is a covering of a connected graph $\mathcal{G}$, then in a sense it *embeds* several copies of $\mathcal{G}$[.]" Together with the Locality Axiom this will allow us to argue that (under certain conditions on the messages) it is indistinguishable for a process whether it is in $\mathcal{G}$ or only in one of the embeddings of $\mathcal{G}$ in $\mathcal{H}$.

Starting with the simple case where $t = 1$ (and thus $n = 3$) we note that a completely connected network has a triangle as network graph (cf. Figure 4.1(a)). Clearly, the hexagonal graph of Figure 4.1(b) provides a covering of that triangle.

Returning to the triangle when we assume that p_0 and p_1 both propose 0 while p_2 behaves Byzantine, by telling p_0 its value is 0 and while it acts as if its value was 1 towards

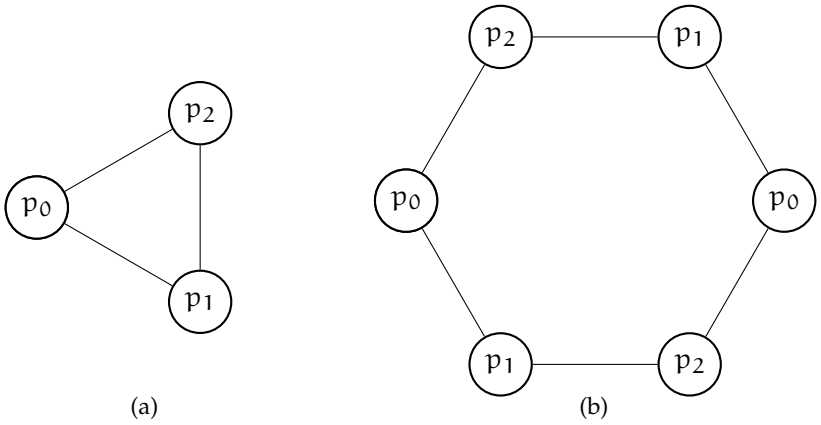


Figure 4.1. *A hexagon covering a triangle.*

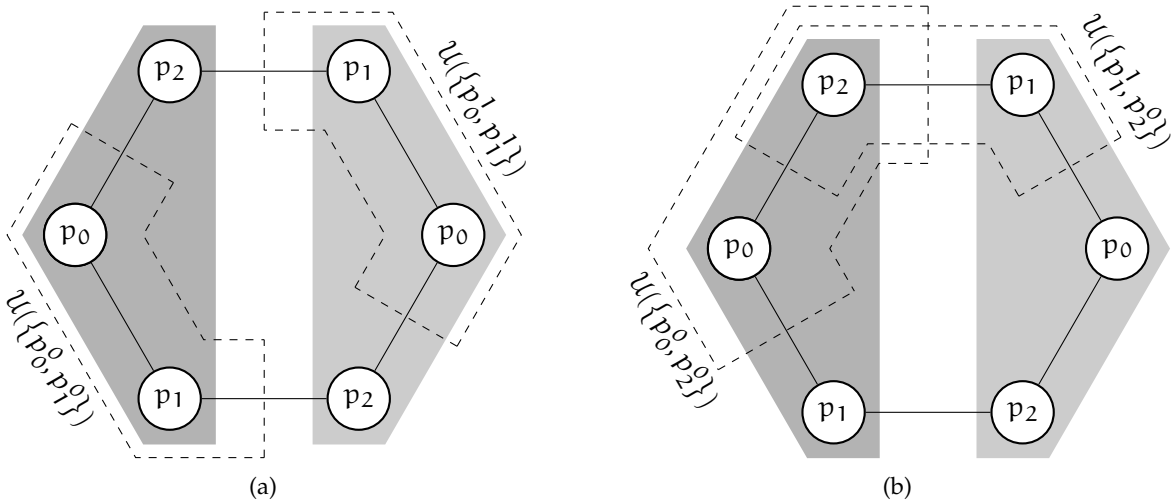


Figure 4.2. *Different subsystems $\mathcal{U}()$ for the three process impossibility proof; Dark gray regions correspond to initial values of* 0*, light gray regions correspond to initial values of* 1*.*

p_1, then the Locality Axiom allows us to conclude that p_0 and p_1 will behave in the same way in the hexagonal graph when the links between each of these two nodes and the respectively neighbouring p_2 carries the same messages as in the three node system. The key to the impossibility result for three processes is to find executions in which this holds (among other conditions). In particular, we are interested in those executions which allow p_2 (or more generally every process) to be correct in the hexagonal graph.

We consider a system of six processes arranged as in Figure 4.2 in which p_0, p_1 and p_2 are the only names of the processes. Moreover, we assume that one set of processes has 0 as their initial value while the others start with their initial value being 1.

In the figure an initial of 0 is indicated by the darker background, but we will use superscripts to indicate the initial value of processes in the text. Now consider the subsystem $\mathcal{U}(\{p_0^0, p_1^0\})$ (Figure 4.2(a)) consisting of those p_0 and p_1 processes which both have 0 as initial value. The same subsystem can also be found in a three process system in which p_2 is the faulty process and behaves in a way such that the links $\langle p_2, p_0 \rangle$

and $\langle p_2, p_1 \rangle$ have (at any time) the same state as in the six process system. Clearly p_0 and p_1 have to reach agreement on 0 by Validity in the three process system, and therefore — that is, by the Locality Axiom — also have to decide on 0 in the six process system. (Since the algorithm we run in the six processes system is not necessarily a consensus algorithm, the term "deciding" just refers to the action assigning the decision value to the output variable here.) The same argument can be used to deduce that the processes in the subsystem $\mathcal{U}(\{p_0^0, p_2^0\})$ (Figure 4.2(b)) decide on 0 as well. Likewise, an equivalent argument can be used for the case where processes start with 1 as their initial value: thus processes in $\mathcal{U}(\{p_0^1, p_1^1\})$ decide on 1 — notwithstanding whether they find themselves[2] in the system of six or only three processes.

As the final subsystem, we consider the subsystem indicated by $\mathcal{U}(\{p_1^1, p_2^0\})$ (Figure 4.2(b)) which corresponds to a three process system where p_0 is faulty. By considering the two singleton subsystems $\mathcal{U}(\{p_1^1\})$ and $\mathcal{U}(\{p_2^0\})$ it turns out that — due to the Locality Axiom — processes have to behave in the same way as in $\mathcal{U}(\{p_0^1, p_1^1\})$ and $\mathcal{U}(\{p_0^0, p_2^0\})$, respectively. This implies both decide on their respective initial values. By virtue of the Locality Axiom the processes must behave in the same way in the three process system corresponding to $\mathcal{U}(\{p_1^1, p_2^0\})$, that is, in a system where p_1 and p_2 are correct while p_0 is faulty and sends the same messages to p_1 and p_2 as the two copies of p_0 in the six process system. Clearly, p_1 deciding on 1 and p_2 deciding on 0 violates the agreement property of consensus, which implies the following Lemma 4.1:

Lemma 4.1. *There is no correct consensus algorithm for a system of three processes, when one may behave Byzantine, even when there are no link failures.*

As a wide range of algorithms demonstrates consensus becomes possible when $t = 1$ and $n > 3$ (cf. Fischer's [1983] survey, or a distributed computing book, e.g., [Lynch 1996; Attiya and Welch 2004]). Most of these algorithms are not restricted to $t = 1$ however, rather it is typically assumed that $t \leq \lfloor \frac{n}{3} \rfloor$. Which leads to the need for a more general lower bound given in Theorem 4.2. The proof is also based on that given by Fischer et al. [1986], which is based on a reduction to the case $n = 3$. Alternatively, one could refer to the direct proof of Pease et al. [1980] (which was done for the interactive consistency problem) .

Theorem 4.2. *There is no correct consensus algorithm for systems of n processes, when t may behave Byzantine and $n \leq 3t$, even if there are no link failures.*

Proof. There are two cases, either $t = 1$, then the Theorem follows from Lemma 4.1. Otherwise when $t > 1$ we assume that there is an algorithm $\mathcal{A}$ which does in fact solve consensus for some $n \leq 3t$.

Again, we consider the three processes p_0, p_1, and p_2 of which one is Byzantine faulty. Clearly this is impossible due to the Lemma 4.1, but we now demonstrate how they can do so based on the assumed algorithm $\mathcal{A}$, which in turn proves that $\mathcal{A}$ cannot exist.

To this end, we divide the set of n processes into three subsets Π_0, Π_1, and Π_2 each consisting of at most t processes. We then let each of these three processes p_i run the algorithm $\mathcal{A}$ for all processes in Π_i. Moreover, we let all the process of Π_i start with p_i's initial value as their initial values. Clearly, for $\mathcal{A}$ there are n processes and at most t processes (that is the processes in one of the three Π_i) are faulty, so $\mathcal{A}$ will ensure that

[2] By this we do not want to indicate they can differentiate between the two cases.

the n processes reach agreement on, say, v. As all simulated processes agree on v it suffices for the three p_i to decide on any simulated processes decision value, to ensure agreement. Validity and termination for the three processes solution follows from the respective properties of $\mathcal{A}$. □

Fischer et al. [1986] also showed a result for the minimal connectivity of systems. In more detail, they showed that, for consensus to be possible, the communication graph has to be at least $2t + 1$-connected. We do not reiterate this proof here, as we are more interested in dynamic faults in this thesis. We do, however, use the technique above to prove our lower bound results for this kind of failures (cf. the next Chapter). As a final note, it should also be mentioned, that Fischer et al. [1986] do not only proved bounds for consensus, but also for clock synchronization and approximate agreement.

4.4. Lower Bounds for Systems with Link Failures

In this section we restrict ourselves to link failures. As noted above, we start out by reiterating a result by Schmid and Weiss [2002] which shows that it is impossible to achieve consensus in a system with four processes, if a single link per round per process may be arbitrarily faulty. In our model this scenario corresponds to $\ell = \ell^a = 1$ and $t = 0$. For the purpose of this proof we do, however, assume that the faulty links (one per process) do not change from round to round. This aspect of the proof actually provides a new insight as Schmid et al. [2009] seem to assume that it is necessary for the faults to hit different links in different rounds, which may be due to the fact that some of their arguments are based on ideas of [Santoro and Widmayer 1989], which need the fact that different links may be hit in different rounds.

In the Figure 4.3 and the following figures used to illustrate the lower bound proofs, we use the following graphical conventions for drawing our network graphs:

Dashed lines indicate links with arbitrary link failures.

Double lines indicate that all processes along a double lined path form a clique, i.e., a fully connected sub graph.

As before (in Section 4.3) the proof is based on assuming the existence of a correct consensus algorithm (for $n = 4$). We consider an execution of this algorithm in a bigger system (of $2n = 8$ processes), which is failure-free but not fully connected. By means of the Locality Axiom we find subsystems for which we can determine the behaviour to from that of systems with $n = 4$, in particular, systems where the decision values are known to be 0 and 1, respectively (due to the Validity property of the assumed algorithm). We then identify a third subsystem intersecting with the previous two, which also corresponds to a legal execution in a system of n processes, for which we can determine the behaviour of processes from the previous two. But since the decision values (of the processes in the first two subsystems) are different we have found an execution which violates Agreement, thus contradicting the assumed existence of a correct algorithm.

This is also the approach taken in the subsequent proofs for larger systems. For those it will become impractical to refer to the subsystems by naming all processes in it (as we have done it in the proof of Lemma 4.1), therefore we introduce more "abstract" names for the three aforementioned subsystems:

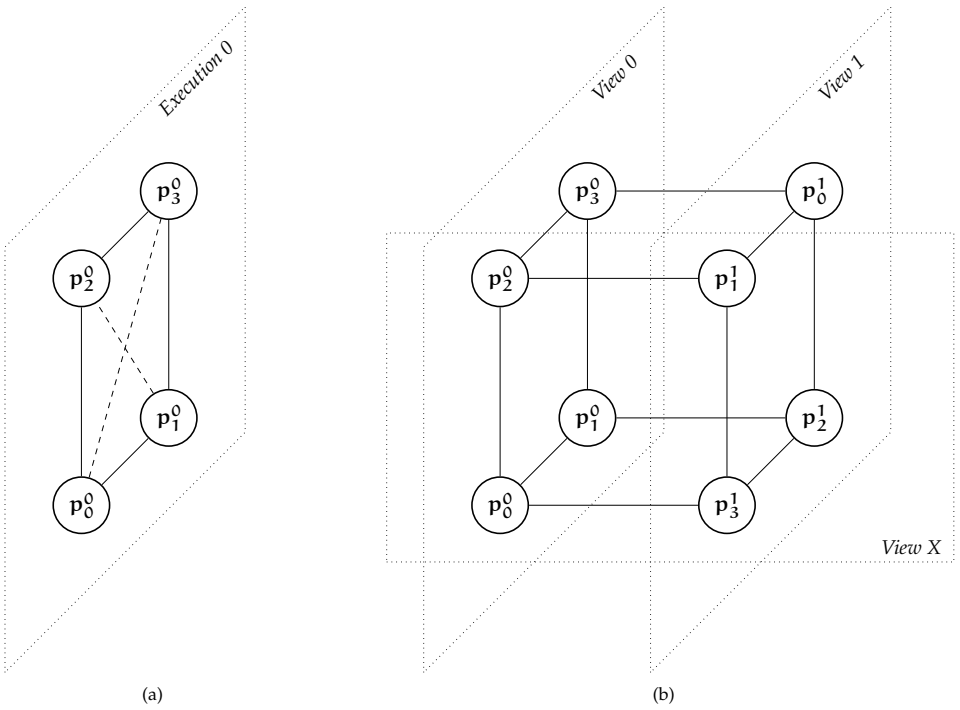


Figure 4.3. *Proving the violation of Agreement in a 4-process system.*

View 0, View 1 and View X. View 0 and View 1 are those subsystems for which the Locality Axiom allows us to argue their indistinguishability from two executions in the n process systems where the validity property forces the decision value to be 0 and 1, respectively. Finally, View X overlaps with the others in the system with $2n$ processes and is indistinguishable from the execution we will refer to as Execution X where Agreement is violated. Here, indistinguishability of a view from an execution is used to denote the fact that due to the Locality Axiom the processes in the view cannot distinguish being executed in the $2n$ system from being executed in a system of n processes.

Theorem 4.3 (Impossibility for $n = 4$)**.** *No deterministic algorithm is able to solve consensus in systems of $n = 4$ processes when a single arbitrary link fault is possible even if there are no process faults, i.e., $t = 0$, $\ell^a = \ell = 1$.*

Proof. We assume that an algorithm $\mathcal{A}$ does exist which is able to solve consensus under the assumptions of the theorem. Then, in any execution where all processes initially agree on some value v they have to decide on that value by the Validity property. Consider for instance executions in the system depicted in Figure 4.3(a). Due to Validity the algorithm has to decide on 0, irrespective of the behaviour of the two faulty links (the ones between p_0^0 and p_3^0 and between p_1^0 and p_2^0). Clearly, the decisions reached are independent of the messages received over the faulty links.

Turning to Figure 4.3(b) we find a system of 8 processes where half the processes (p_i^0) start with 0 and the other half (p_j^1) starts with 1 as their initial value. The neighbourhood of each process within View 0 is the same as before in Execution 0. Since we have established that the messages received over the faulty links in Figure 4.3(a) do not influence the behaviour of processes, we can conclude (due to the Locality Axiom) that the pro-

cesses p_0^0, p_1^0, p_2^0, and p_3^0 will behave in the same way in View 0 of Figure 4.3(b) as in one of the executions in the system of Figure 4.3(a). We call this execution, *Execution* 0. Likewise one can find *Execution* 1, which has to decide on 1 and is locally indistinguishable (to p_0^1, p_1^1, p_2^1, and p_3^1) from being executed in View 1 in Figure 4.3(b).

To complete the proof we consider a third subsystem, i.e., the View X in Figure 4.3(b). If we can find a legal execution in a 4 processes system in which processes behave in the same way as in the execution of Figure 4.3(b), then we deduce that Agreement has not been reached (and thus $\mathcal{A}$ is not correct), as, e.g., p_0^0 decides on 0 while p_1^1 decides on 1.

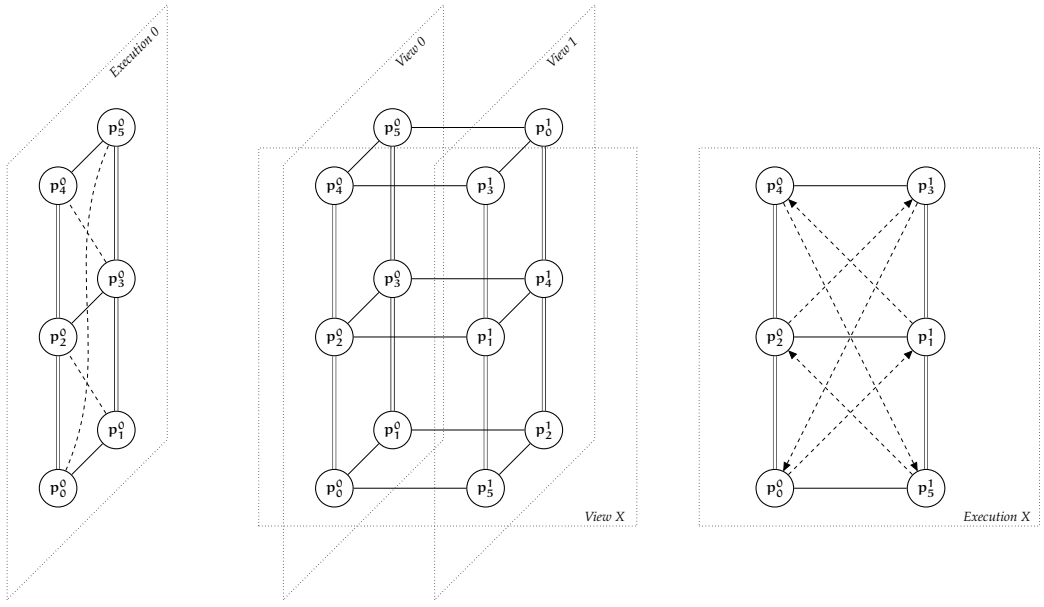
Using the Locality Axiom, we can indeed find such execution. All that is required is to consider a system of four processes named p_0, p_1, p_2, and p_3 in which the links $\langle p_0, p_1\rangle$ and $\langle p_2, p_3\rangle$ are faulty. Moreover, we let p_0 and p_2 start with 0, while p_1 and p_3 have 1 as their initial value. The sought after execution is the one, where the arbitrary faulty links deliver exactly the same messages as the corresponding (correct) links in Figure 4.3(b). □

Using the same technique as for generalizing Lemma 4.1 to Theorem 4.2 we could easily be extend the above theorem to hold for arbitrary numbers of link failures (that is, $\ell^a \geq 1$) to reach an impossibility for $n = 4\ell^a$ processes. As our aim is to prove a bound for lossy and arbitrary faulty links (and eventually Byzantine processes), we do not take this path. Instead, we will use the same strategy as in the proof of Theorem 4.3. To this end, we will now review some of the steps of the proof, and introduce notions that will be handy in obtaining the general result.

First, we recall that we have used two kinds of systems one of $n = 4$ processes and one with $2n = 8$ processes. Both systems were based on specific communication graphs. In the following we will denote communication graphs of the n process systems by γ and the communication graph of the $2n$ process system by Γ. As above, γ will cover three different graphs (γ_0, γ_1, and γ_X) for the n process systems, where γ_E is used as communication graph for Execution E ($E \in \{0, 1, X\}$).

In order to define the connectivity in these graphs (and to argue why it Γ graph covers the γ graphs), we will define the neighbourhood of each process in Γ. We will define the neighbourhood of each process p, by three disjoint sets $\mathcal{W}(p)$, $\mathcal{S}(p)$, and $\mathcal{O}(p)$. The meaning behind these three sets is that neighbours in $\mathcal{W}(p)$ will always be *within* the same View as p; neighbours in $\mathcal{S}(p)$ are those neighbours (aside from those in $\mathcal{W}(p)$) which will have the *same* initial values, while those neighbours in $\mathcal{O}(p)$ have the other initial values in the execution in Γ. Returning to 4.3(b), we would thus have $\mathcal{W}(p_0^0) = \{p_2^0\}$, $\mathcal{S}(p_0^0) = \{p_1^0\}$, and $\mathcal{O}(p_0^0) = \{p_3^1\}$. Recall that when arguing about the behaviour of processes in View 0 and Execution 0 (that is, when arguing about Γ covering the γ_0) we noted that in Execution 0 the link to p_3^0 was to be considered faulty. In the following proofs we will again follow this pattern. That is, when arguing about Γ covering the γ in which Executions 0 and 1 are run (i.e., γ_0 and γ_1), we will consider the links between p and the processes in (p) to be faulty. When arguing that Γ also covers the γ_X in which the third execution (which we will call Execution X in the future) is run, we will consider the links from p to $\mathcal{S}(p)$ to be faulty instead. (Revisiting the proof of Theorem 4.3, we see that we have already done this when considering the link between p_0^0 and p_1^0 to be faulty in the execution leading to the contradiction.)

In all the γ graphs, the neighbours of p connected by faulty links, will have the opposite (with respect to the respective neighbour in Γ) initial value. (Cp. p_0^0's neighbour



(a) (b) (c)

Figure 4.4. *Proving the 6-process impossibility.*

p_3^0 resp. p_3^1 in Figure 4.3(a) resp. 4.3(b).) We denote the opposite value of v by $\overline{v} := 1 - v$ and for the neighbour set $\mathcal{O}$, we define $\overline{\mathcal{O}(p)} := \{p_i^{\overline{v}} : p_i^v \in \mathcal{O}(p)\}$ the set of neighbours connected via faulty links in the γ_0 and γ_1. That is, in γ_0 and γ_1 process p is connected to $\mathcal{W}(p)$ and $\mathcal{S}(p)$ via correct links and to processes in $\overline{\mathcal{O}}(p)$ via arbitrary faulty links. $\overline{\mathcal{S}(p)}$ is defined and used analogously for γ_X.

We now introduce the general definitions of the three sets of neighbours for any process p_i^v. Before showing the general result, we will prove a impossibility result for 6 processes (where $\ell = 2$, $\ell^a = 1$, and $t = 0$). Note that, the calculations involved in determining the neighbour sets $\mathcal{W}$, $\mathcal{S}$, and $\mathcal{O}$ using the following definitions are all done modulo n.

$$\begin{aligned}
\mathcal{W}(p_i^v) &= \{p_j^v : \exists k \in \mathbb{N},\ j = i + 2k\} \\
\mathcal{S}(p_i^v) &= \left\{p_j^v : \exists k \in \mathbb{N},\ (k < \ell^a) \wedge \left(j = i + (-1)^i(1 + 2k + 2v\ell^o)\right)\right\} \\
\mathcal{O}(p_i^v) &= \left\{p_j^{\overline{v}} : \exists k \in \mathbb{N},\ (k < \ell^a) \wedge \left(j = i - (-1)^i(1 + 2k)\right)\right\}
\end{aligned}$$

Applying these definitions to the aforementioned case of $n = 6$ processes, $\ell = 2$ faulty links per process of which $\ell^a = 1$ may behave arbitrarily, and $t = 0$ faulty processes, one obtains the Γ of Figure 4.4(b).

Theorem 4.4. *There is no deterministic algorithm that solves consensus for a single arbitrary link fault and a single omissive link fault in systems with $n = 6$ even when all processes are correct, i.e., $\ell = 2, \ell^a = 1$ and $t = 0$.*

Proof. Consider the network of Figure 4.4(a), where link faults always occur on the same links. Recall that, dashed lines indicate arbitrary link faults, and total omissions occur on the links missing in the figure (that is, $\langle p_0^0, p_3^0\rangle$, $\langle p_2^0, p_5^0\rangle$, and $\langle p_1^0, p_4^0\rangle$). Since all processes

start with the same value 0, all correct consensus algorithms must ensure that in all runs in this system all processes decide on 0 due to the Validity property.

Again we let the same algorithm run in Γ (Figure 4.4(b)) and obtain specific behaviours for links in the $\mathcal{O}$ set of processes p_i^0 (View 0). Clearly, there is one specific execution (called Execution 0) in γ_0 in which the links to neighbours in $\overline{\mathcal{O}(p_i^0)}$ behave in exactly the same way. Now, the Locality Axiom — once again — allows us to conclude that all processes in View 0 also "decide" on 0. An analogous argument can be made about processes deciding 1 in Execution 1 in γ_1 and their behaviour in View 1 (i.e., the processes p_i^1 in Figure 4.4(b)).

As a final step, we observe that the processes of View X (see Figure 4.4(b)) cannot distinguish being in View X from the execution in γ_X (Figure 4.4(c)). Therefore, they still have to behave in the same way and thus decide on their respective initial values in the execution in γ_X. However, these decisions violate the Agreement property of the assumed algorithm. Thus, no such algorithm can exist. □

Note that, in contrast to the case of 4 processes, the faulty links are not necessarily symmetric in the general case. This can be seen in Execution X (Figure 4.4(c)). As an example consider the links of p_2^0, while the link to p_3^1 is arbitrary faulty, the link in the opposite direction is omissive.

In the case of 6 processes we could draw the communication graphs resulting from the definitions of the neighbor sets. In general this is not possible, therefore we have to prove some characteristics of these sets in order to know we can use them in our proofs: In particular, the three sets have to be disjoint (with respect to the identifiers only, i.e., ignoring the initial values). This is due to the fact that a process should only have one neighbour with a certain identifier. Moreover, the three sets have to respect the size requirements as defined by the semantics of ℓ^o, ℓ^a, ℓ, and n. We will show these properties in the following Lemma 4.5. However, satisfying the size requirements is only sufficient for showing that no process has more than ℓ outgoing faulty links (and no more than ℓ^a arbitrary faulty outgoing links). It is not sufficient for proving the bounds hold on inbound links as well. To this end, we show (in Lemma 4.6) that the relations induced by the sets are symmetric.

Lemma 4.5. *The three sets $\mathcal{W}(p_i^v)$, $\mathcal{S}(p_i^v)$, and $\mathcal{O}(p_i^v)$ that make up the neighbors of p_i, are disjoint sets of sizes $\ell^o + 2\ell^a$, ℓ^a and ℓ^a respectively, when the number of nodes is $n = 2\ell^o + 4\ell^a$.*

Proof. We show the statement about the sizes first: Since n is even by definition, set $\mathcal{W}$ will contain all even or uneven identifiers smaller than n, depending on whether i itself is even or not. That is, the set $\mathcal{W}$ will contain $\{0, 2, \ldots, n-2\}$ or $\{1, 3, \ldots, n-1\}$, respectively, each of which has exactly $\frac{n}{2} = \ell^o + 2\ell^a$ members. The sets $\mathcal{O}$ and $\mathcal{S}$ are obviously of size ℓ^a, since $k \in \mathbb{N}$ and $k < \ell^a$.

Since the term added (or subtracted) from i to determine $\mathcal{S}(p_i^v)$ and $\mathcal{O}(p_i^v)$ is always odd, these two sets cannot share any members with $\mathcal{W}(p_i^v)$. To show that $\mathcal{S}$ and $\mathcal{O}$ do not overlap, we assume the contrary, that is we assume that there are some $0 \leq k, k' < \ell^a$ that map i to the same value:

$$i + (-1)^i(1 + 2k + 2v\ell^o) = i - (-1)^i(1 + 2k') \pmod{n}$$

By eliminating i and $(-1)^i$ we obtain $2 + 2k + 2k' + 2v\ell^o = 0 \pmod{n}$. There is, however, no solution of this equation since

$$0 < 2 + 2k + 2k' + 2v\ell^o \leq 2 + 4(\ell^a - 1) + 2v\ell^o < 4\ell^a + 2\ell^o = n.$$

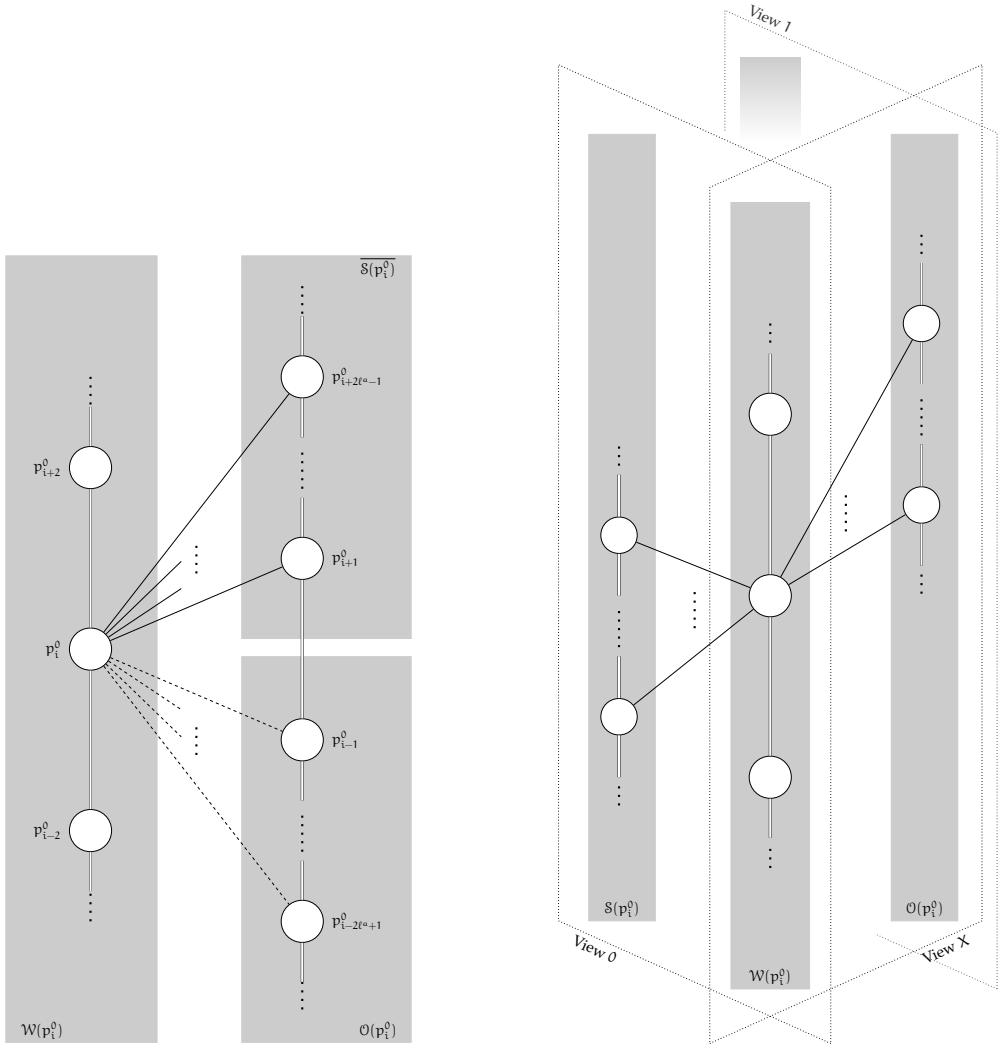


Figure 4.5. *Neighbors of* p_i^0 *in (a)* γ_0 *and (b)* Γ

Therefore, $\mathcal{S}$ and $\mathcal{O}$ cannot overlap. □

Lemma 4.6. *Given the definitions of* $\mathcal{S}$ *and* $\mathcal{O}$ *above the relations induced by these sets are symmetric, i.e.,* $\forall \mathcal{X} \in \{\mathcal{W}, \mathcal{S}, \mathcal{O}\} : p_j^w \in \mathcal{X}(p_i^v) \Rightarrow p_i^v \in \mathcal{X}\left(p_j^w\right)$.

Proof. Since $\mathcal{W}(p_i^v)$ contains all the other processes with odd resp. even identifiers and the same initial value it is obvious that $\mathcal{W}$ is indeed symmetric.

For the symmetry of $\mathcal{S}(p_i^v)$ and $\mathcal{O}(p_i^v)$ we observe that for all identifiers j in these sets it holds that $(j \bmod 2) = 1 - (i \bmod 2)$ and thus $(-1)(-1)^i = (-1)^j$. Therefore, assuming

that $p_j^v \in \mathcal{S}(p_i^v)$ we have

$$j = i + (-1)^i(1 + 2k + 2v\ell^o) \pmod{n},$$

which leads to

$$-i = -j + (-1)^i(1 + 2k + 2v\ell^o) \pmod{n},$$

multiplying by (-1) results in

$$i = j + (-1)(-1)^i(1 + 2k + 2v\ell^o) \pmod{n},$$

and finally we obtain

$$i = j + (-1)^j(1 + 2k + 2v\ell^o) \pmod{n},$$

from which it follows that $p_i^v \in \mathcal{S}(p_j^v)$, since v all processes in $\mathcal{S}(p_i^v)$ have the same initial value tag v.

The proof for the symmetry of $\mathcal{O}$ proceeds in the same way, by showing that $j = i - (-1)^i(1 + 2k) \pmod{n} \Leftrightarrow i = j - (-1)^j(1 + 2k) \pmod{n}$ and observing that when $p_j^{\overline{v}} \in \mathcal{O}(p_i^v)$, all members of $\mathcal{O}(p_j^{\overline{v}})$ have the initial value $\overline{\overline{v}} = 1 - (1 - v) = v$ □

The two lemmas above allow us to use the same techniques as above to prove the impossibility of consensus, but without a fixed number of processes. Once again the idea is to assume an algorithm exists and to find subsystems that correspond to executions where the algorithm will violate Agreement when we assume that it achieves Validity.

Theorem 4.7. *There is no deterministic algorithm that solves consensus in a system with* $n = 2\ell^o + 4\ell^a$ *processes with up to* ℓ^a *arbitrary and* ℓ^o *omissive link faults per process per round and no process failures.*

Proof. Suppose that there exists a distributed algorithm $\mathcal{A}$ that solves consensus under our system model with $n = 2\ell^o + 4\ell^a$. As before, we consider a system Γ with $2n$ processes and determine the "decision values" in this system by finding systems of n processes that have communication graphs γ_0, γ_1, and γ_X, which are all covered by Γ. Thus the Locality Axiom allows us to deduce that the decision values of processes in γ are the same as in these n-process systems.

As a first step we determine the decision value of the processes p_i^0. To do so, we consider the n process system in which all processes have the initial value 0 and which uses the communication graph defined by γ_0 (Figure 4.5(a)). Thus the links from p_i^0 to all processes in $\mathcal{W}(p_i^v) \cup \mathcal{S}(p_i^v)$ are non-faulty, while those to processes in $\overline{\mathcal{O}(p_i^0)}$ are arbitrary faulty. (The links to the remaining processes are assumed to be purely omissive.) By Lemmas 4.5 and 4.6 this system abides to our failure assumptions, and due to the Validity of the assumed algorithm the decision value of all executions in this system must be 0. One specific execution in γ_0 is the one where the arbitrary faulty links carry exactly the same messages as in Γ (Figure 4.5(b)), thus the processes p_i^0 all decide on 0 in Γ. Analogously, we can determine that the decision value of the processes p_i^1 in Γ must be 1.

We now consider a third execution (referred to as Execution X) of n processes, where half the processes have 0 and the other half has 1 as their initial value and the messages received are exactly the same as in Executions 0 and 1. In the $2n$ system this execution corresponds to the processes $\{p_{2k}^0\} \cup \{p_{2k+1}^1\}$ with $k < \frac{n}{2}$. These processes are collectively

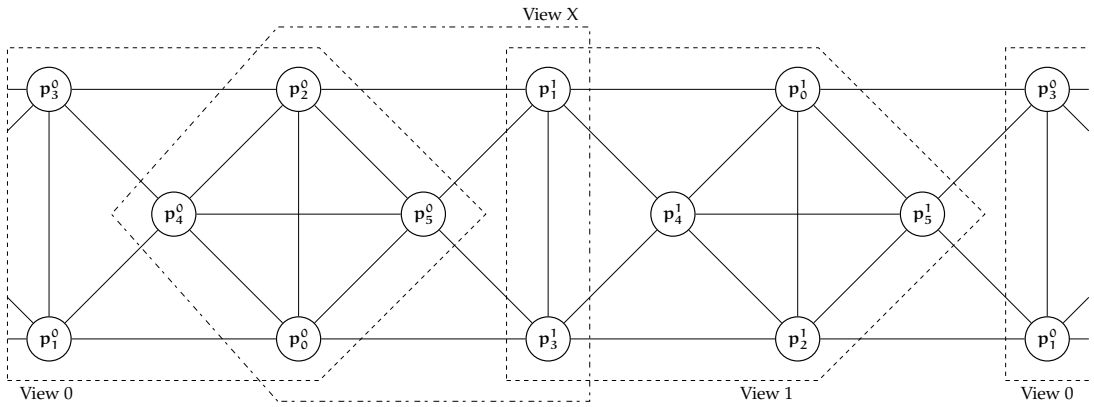


Figure 4.6. *Drawing of an unrolled cube. (Nodes p_3^0 and p_1^0 are shown twice.)*

referred to as the processes in View X. Recall that, in γ_X the links to processes in $\overline{\mathcal{S}}$ are those that are arbitrary faulty, while those to processes in $\mathcal{O}$ (and $\mathcal{W}$) are considered non-faulty.

In the execution in the system defined by γ_X the decision value of processes in the subsystem consisting of the processes $\{p_{2k}^0\}$ must be 0 due to the Locality Axiom and the arguments about View 0 above. The processes in $\{p_{2k+1}^1\}$ likewise still decide on 1. However, this behaviour violates Agreement in Execution X, thereby establishing the required contradiction. □

Note that this result also holds for the case where $\ell^a = 0$, that is there are only $\ell = \ell^o$ omissive links per round per process. This is a somewhat paradoxical case, however, as $\forall p \in \Pi : \mathcal{S}(p) = \mathcal{O}(p) = \emptyset$ when $\ell^a = 0$. In other words, Γ and the γs are partitioned into cliques of size $\frac{n}{2} = \ell$.

4.5. Unifying Link and Process faults

As the final lower bound of this section, we will show that in the presence of link faults (as defined by our system model) it is impossible to reach consensus among $n \leq 2\ell^o + 4\ell^a + 2t$ processes. On the one hand if $t > 2\ell^o + 4\ell^a$, the bound trivially follows from the $n > 3t$ lower bound given in Theorem 4.2. On the other hand, if link faults become too prevalent more than $3t$ correct processes are be needed. We will show that this is necessary by extending Theorem 4.7 to show the aforementioned $n \leq 2\ell^o + 4\ell^a + 2t$ bound. Therefore, in general, there must be $n > 2t + max\{t, 2\ell + 2\ell^a\}$ processes to allow reaching consensus. Algorithm 5.1 and its proof of correctness will show that $n > 2t + max\{t, 2\ell + 2\ell^a\}$ is indeed sufficient.

Before considering the general case, we will first consider a simple extension of Theorem 4.3. More specifically, we examine the case where (at most) one process is faulty and every process has one arbitrary faulty incoming/outgoing link. Thus, we have that $t = 1$ and $\ell = \ell^a = 1$, which leads to $n = 6$. To arrive at this number of processes, we extend the cube of Theorem 4.3 by adding one process to each face. The definition of a view is changed such that each view now comprises 6 (instead of 4) nodes. Figure 4.6 shows an "unrolled" cube with these extra processes, and how views are defined here.

Examining Figure 4.6 in more detail, we observe that in the executions corresponding

to Views 0 and 1, the link between p_0 and p_3 as well as the link between p_1 and p_2 must be considered faulty. Nodes p_4 and p_5 have 5 correct links each, but the processor p_5 is considered faulty.

As before, all processes in View 0 use $v = 0$ and therefore must decide 0 by the Validity property for the consensus algorithm running in the corresponding 6-process system. Similarly, in View 1, all processes must decide 1. Once again the processes in View X face a problem: Since for example process p_0^0 observes exactly the same messages in View X as in View 0, it must decide 0 as before. Similarly, as process p_3^1 observes exactly the same messages in View X as in View 1, it must decide 1. However, these decisions violate Agreement in the 6-process system corresponding to View X, thereby establishing the required contradiction for completing the proof of Theorem 4.8.

Theorem 4.8. *There is no deterministic algorithm that solves consensus with one arbitrary link fault ($\ell = \ell^a = 1$) and one Byzantine process ($t = 1$) fault in a system with $n = 6$ processes.*

Turning to the general case, we first show that $n > 2t + 2\ell + 2\ell^a$ is needed to achieve consensus, and then combine this result with the lower bound of Pease, Shostak, and Lamport [Pease et al. 1980] to arrive at our final result, stated in Theorem 4.10 below.

Theorem 4.9. *There is no deterministic algorithm that solves consensus for ℓ^a arbitrary link faults, ℓ^o purely omissive link faults and t arbitrary process faults in a system with $n = 2t + 2\ell + 2\ell^a$ processes.*

Proof. Suppose that there is an n processes execute a distributed algorithm $\mathcal{A}$ which solves consensus under our assumptions. In order to derive a contradiction, we once again arrange $2n$ processes in a graph, denoted Γ, as follows: First we build the graph exactly as described for the $t = 0$ case in Section 4.4, that is, for $2n' = 2(2\ell + 2\ell^a)$ processes. Then we add for each v two sets of processes $\mathcal{E}[v] = \{p_i^v \mid n' \leq i < n' + t\}$ and $\mathcal{F}[v] = \{p_i^v \mid n' + t \leq i < n' + 2t\}$. (These sets play the roles of p_4^v and p_5^v in the argument for Theorem 4.8.) Processes in $\mathcal{E}[v]$ are connected to all processes in View v, that is, all p_i^v with $0 \leq i < n$. Processes in $\mathcal{F}[v]$ are connected to all processes in View v except those in $\{p_i^v \mid i \bmod 2 = 1 \wedge i < n'\}$. Instead, we connect processes in $\mathcal{F}[v]$ to $\left\{p_i^{1-v} \mid i \bmod 2 = 1 \wedge i < n'\right\}$. Thus, each process in $\mathcal{F}[v]$ has connections to processes in the set $\{p_i^w \mid (0 \leq i < n') \wedge (w = v + i \bmod 2)\} \cup \mathcal{E}[v] \cup \mathcal{F}[v]$. In summary, we add two sets (to each view) of processes instead of only two processes as in Figure 4.6 and connect them analogously. We observe that when adding the $2t$ processes, we do not add any more faulty links to the executions corresponding to any of the views and therefore Lemmas 4.5 and 4.6 still apply here. Note that in the executions corresponding to the views 0 and 1 all processes in the $\mathcal{E}$ sets are correct, while those in the $\mathcal{F}$ sets behave faulty. This concludes the construction of our $2n$ process system, with views 0 and 1, each of which corresponds to an execution of $n = 2t + 2\ell + 2\ell^a$ processes.

Now — once again — for $v \in \{0, 1\}$, consider the processes in Views v, which all have initial value v, respectively. By applying the Validity property of $\mathcal{A}$ to the executions in γ_v all processes must decide on v. To reach the desired contradiction, we now have to choose n processes in Γ, such that their perceptions and behaviour correspond to a valid Execution X in the n process system whose communication graph is γ_X. This can be

achieved by selecting the processes in

$$\mathcal{X} \quad = \mathcal{E}[0] \ \cup \ \mathcal{F}[0] \cup \quad \left\{ p_i^0 \mid (i \bmod 2 = 0) \wedge (i < n') \right\} \ \cup \\ \left\{ p_i^1 \mid (i \bmod 2 = 1) \wedge (i < n') \right\}.$$

Since there are processes with different initial values in $\mathcal{X}$ and all processes have to decide on their initial value by the construction above, we have reached the contradiction if we can show that $\mathcal{X}$ can be mapped to a valid γ_X. This is done in two steps: We first show that (i) $\mathcal{X}$ is indeed a set of n distinct processes, and (ii) that our failure assumptions are not violated in $\mathcal{X}$.

As for (i) we observe that the sets that make up $\mathcal{X}$ are all non-overlapping with respect to their identifiers. The sets $\mathcal{E}[0]$ and $\mathcal{F}[0]$ are of size t each, while the last two components are of size $\frac{n'}{2}$ each. This adds up to $n' + 2t = n$ processes. As for (ii) we observe that in View X (in contrast to the Views ν) processes in $\mathcal{F}[0]$ are perceived in the same way by all processes. That is, they can be counted as correct processes with correct links only. In Execution X the t processes in $\mathcal{E}[0]$ have to be considered Byzantine, since the asymmetric behaviour[3] of those processes exceeds what could be attributed to link failures. As we can blame all asymmetric behaviour on the processes being faulty, the outgoing links of $\mathcal{E}[0]$ do not count towards the ℓ^a limit on the incoming links of neighbors. As for the incoming links of $\mathcal{E}[0]$, we note that we can do not have to check the link failure bounds as these apply only to correct processes. For the other two subsets of $\mathcal{X}$ we observe that all processes are correct, and that these processes alone correspond to a valid View X in a $2n'$ system (as in Lemma 4.5). Therefore (ii) holds as well. □

What remains, is to relate the result of Theorem 4.9 with the result of Theorem 4.2. Obviously, the resilience-bounds which can be deduced from the two theorem must both be guaranteed in order to have a system in which consensus is solvable. Or the other way round, when one of them is not fulfilled consensus remains unsolvable. Thus we conclude this chapter with the observation that,

Theorem 4.10. *There is no deterministic algorithm that solves consensus with ℓ^a arbitrary link faults, ℓ^o purely omissive link faults and t arbitrary process faults in a system with $n \leq 2t + \max\{t, 2\ell + 2\ell^a\}$ processes.*

At this point the question of how the results of this chapter relate to other impossibility results for link failures naturally arises. One point that is interesting is that while we have assumed that the faulty links may change from round to round, this fact is not used in the proofs. This is in contrast to Santoro and Widmayer [1989], who assume that the transmission failures may occur in a block around different processes in different rounds. (Informally their impossibility result is thus based on a similar construction as that of Fischer et al. [1985], where the process unable to communicate "in time" to others changes over time as well.) Moreover, Santoro and Widmayer require only $n-1$ omissions or $\lceil \frac{n}{2} \rceil$ Byzantine faulty links system-wide in each round, whereas $\ell > \frac{n}{2}$ omissions or $\ell > \frac{n}{4}$ Byzantine faulty links are required per process for our construction to work. Therefore, none of the two results is more general.

[3] This asymmetric behaviour is caused by some processes receiving the same messages from $\mathcal{E}[0]$ in Execution X as they received from $\mathcal{E}[0]$ in Γ, while others receive the same messages as they received from $\mathcal{E}[1]$ in Γ.

PART III

ALGORITHMS

CHAPTER 5

RETRANSMISSION PROTOCOLS

IN THE PREVIOUS Chapter we proved a lower bound on the number of processes required to solve consensus in the presence of Byzantine processes and a certain class of moving link failures (cf. Theorem 4.10). This naturally leads to the question whether the given bound is tight. Instead of finding a matching consensus algorithm, we will show how to mask these link failures in an optimal way. By combining an (optimal) reliable channel simulation with an optimal consensus algorithm, that requires reliable links, we obtain an optimal solution for consensus with respect to resilience. When we talk about channels in the following, we mean simulated entities connecting processes over the (physical) links. Similar to reliable links (cf. Chapter 2), a channel is reliable when every message that is sent over the channel by a non-faulty process is received exactly once (at any non-faulty receiver). This definition assumes uniquely identifiable messages, as when two messages with the same content are sent, each of the two (identical) messages is required to be delivered once. Moreover, messages from correct processes are delivered only if the message was really sent by that correct process.

Our implementation of reliable channels will use two rounds of communication thus doubling the termination time of any algorithm (and increasing the message load n-fold). It is similar to the simulated authentication algorithms of Srikanth and Toueg [1987a], which has been used to solve a wide range of agreement problems [Srikanth and Toueg 1987a; Toueg et al. 1987; Srikanth and Toueg 1987b; Dwork et al. 1988]. In Section 5.2, we show how the properties of both reliable channels and simulated authenticated messages can be achieved in three rounds per broadcast (instead of four as one would need when simply plugging the two simulations together. Note that these are simulations in the sense of Chapter 12 in the book by Attiya and Welch [2004] and the work by Neiger and Toueg [1990] and not algorithm transformations (recall Chapter 3), with the most prominent difference being that with simulations one can use multiple independent threads of control as one is not bound to providing replacements for operations.

We use the same model as in the previous chapter for our analysis. Using this very restrictive model has the advantage that we can focus on how failures are masked in the algorithms and proofs. That is, we assume a system with a communication-closed round

structure, up to t faulty processes and up to ℓ incoming/outgoing faulty links per round and per process (of which ℓ^a may be value faulty). Note, however, that the round models we implement are communication-open when there are Byzantine failures. Without such failures there the resulting round structure would be communication-closed as well. This is due to the fact that only faulty processes can cause enough asymmetry for processes to deliver a message in different rounds.

5.1. Masking Link Failures

In general a channel from a sending process to a set of receivers is reliable, if it guarantees the following properties with respect to some message *msg*:

Channel Validity: If a correct process sends *msg*, then all correct receivers eventually deliver *msg*.

Channel Integrity: Every correct receiver delivers *msg* at most once, and only if some process actually sent *msg*.

Our definition of a reliable channel is equivalent to the Validity and Integrity properties of a reliable broadcast as defined by Hadzilacos and Toueg in their chapter on broadcasting in Mullender's book [1993]. Besides the two properties above they also introduce a property called Agreement, which — in short — requires relaying (cp. the properties discussed in Section 5.2). Omitting this property has no impact as long as senders and receivers are correct. Link faults that occur on links where either the sender or the receiver is faulty may still exhibit arbitrary behavior/message loss, but these effects can safely be attributed to the faulty process.[1] Consequently, our channel does not guarantee that all receivers agree on the delivered value if the sender was faulty, thus contrasting reliable broadcast [Hadzilacos and Toueg 1993].

As mentioned before, the definition of Channel Integrity implies that all messages are unique. Often it is reasonable to assume that all messages are unique, as it can be enforced by using unique sequence numbers for every message. Since the link failures we would like to tolerate are defined for round-based systems only, we do not keep sequence numbers explicitly, but rather use round numbers.

As our reliable channel implementation (and the authenticated channels presented in Section 5.2) are based on rounds, and the algorithms using them are so as well, we need to find a concept that allows us to discriminate between the two. Thus, we will in this chapter refer to micro- and macro-rounds. The micro-rounds are what our channel implementations use, while the algorithms built atop these channels will use macro-rounds. A macro-round ends after k micro-rounds. (For our reliable channel implementation $k = 2$.) That is, the code of the macro-round ρ is executed when the (micro-) round counter reaches $k\rho$.

For simplicity, we assume that the algorithm using our channel implementation is broadcast based. In the above reliable channel properties there is no notion of rounds, as we are concerned about round-based systems, we prove the following variant of Channel Validity, which is stronger in such systems (but, as it refer to rounds, not as general).

[1] This is due to the fact that links from benign faulty processes can only loose messages, while arbitrary link failures are only possible with Byzantine processes.

```
1:  variables
2:      r ∈ ℕ                                                    // the current macro-round
3:      ∀q ∈ Π,  delivered[q] ← ∅        // the set of rounds for which we have delivered messages from q

4:  procedure reliably-send(msg)
5:      send(INIT, p, msg, r) to all nodes [including itself]

6:  thread reliably-receive :
7:      in each micro-round:
8:      if received(INIT, q, msg, k) from q in micro-round 2k then
9:          send(ECHO, q, msg, k) to all nodes [including itself]
10:     if received(ECHO, q, msg, k) from n − t − 2ℓ distinct processes in one micro-round then
11:         if k ∉ delivered[q] then
12:             delivered[q] = delivered[q] ∪ k
13:             reliably-deliver(msg, q, k)
```

Algorithm 5.1. *Reliable Channel Implementation, code for process* p

Moreover, as a process may send the same message in multiple rounds, we prove a variant of Channel Integrity which captures this fact.

Channel Validity': If a correct process sends *msg* in macro-round r, then all processes deliver *msg* in macro-round r.

Channel Integrity': Every correct receiver delivers *msg* only for those rounds in which it was sent *msg*.

Our reliable channel implementation (Algorithm 5.1) proceeds as follows: To reliably send a message on the channel to a receiver, the sending process uses the *reliably-send* function (line 4 ff.). The *reliably-receive* thread runs at all processes and forwards the message to the designated receiver(s), where the thread will also deliver it to the next protocol level through *reliably-deliver*.

One particularly important detail of Algorithm 5.1 (and Algorithm 5.2) is that it resends all messages in every round and only uses messages received in the current round in its rules. This is necessary, because link failures can produce ℓ^a erroneous ECHO-messages per round at any correct process, which could otherwise accumulate over multiple rounds to surpass any threshold.

Theorem 5.1. *Algorithm 5.1 guarantees a reliable channel between each correct sender and all correct receivers, if* $n > 2t + 2\ell + 2\ell^a$.

Proof. We denote the sending process by s. We show each of the two properties separately, splitting the proof into two parts:
Channel Integrity': A correct process only delivers a messages if it has received at least $n - t - 2\ell$ copies of a ECHO-message from distinct processes. We observe that since $n > 2t + 2\ell + 2\ell^a$ this corresponds to at least $t + 2\ell^a + 1$ echo messages corresponding to the message to be delivered.

Recall that (i) at most t processes are arbitrary faulty, (ii) at most ℓ^a (correct) processes can receive an INIT-message from s out of thin air (that is, without s sending such message) and thus respond with corresponding ECHO-messages, and (iii) each receiver can receive at most ℓ^a ECHO-messages from processes, which never sent them. This results in at most $t + 2\ell^a < n - t - 2\ell$ ECHO-messages which can be received by some receiver

if s has not sent an INIT-message. Therefore it is clear that a process will only deliver a message if the sender has sent the INIT-message. As the original macro-round number is attached to all INIT- and ECHO-messages, a message clearly is only delivered for a round, for which it was sent. Moreover, adding the round number to the set *delivered*[s], when a message from s is delivered, ensures that the message is delivered only once.
Channel Validity: Since — by assumption — s is correct, it sends the same message to all processes $p \in \Pi$. There are at least $n - t - \ell$ correct processes that receive the message s has sent and will therefore forward corresponding ECHO-messages to the all processes (line 9). Thus, each correct process will receive at least $(n - t - \ell) - \ell = n - t - 2\ell$ and will therefore, execute the code following line 10. This takes exactly two micro-rounds and thus one macro-round. □

From this theorem it follows that by using our reliable channel implementation (Algorithm 5.1) for communication and any consensus algorithm that requires $n > 3t$ (examples include the work by Pease et al. [1980], Dwork et al. [1988], Garay and Moses [1993], and Dolev et al. [1982]) one can solve consensus in the presence of link faults in accordance to our assumptions with $n > 2t + max\{t, 2\ell + 2\ell^a\}$. These possibility results indicate that the lower bound of Theorem 4.10 is tight; respectively the impossibility of Theorem 4.10 indicates that these consensus solutions are optimal with respect to their resilience.

5.2. Authenticated Channels

In this section we turn our attention from simulating reliable channels to simulating authentication. One way to implement authenticated channels is by means of signed messages (e.g., [Rivest et al. 1978; NIST 1992]). This leads to problems, however, when one considers Byzantine faults. In general Byzantine faults are assumed to be able to send messages with *any* content. In the context of cryptography based authentication this is no longer true, as in order to make signatures useful Byzantine processes must be assumed to be unable to (erroneously) send messages containing data with valid signatures of other processes (unless the other process has signed that data before). Such an assumption clearly precludes finding a simple formal definition of Byzantine behaviour in models with authentication [Lynch 1996, page 115]. Moreover, cryptography involves computationally expensive operations, which cannot always be afforded in power-limited systems.

This makes the non-authenticated implementation of the broadcast primitive developed by Srikanth and Toueg [1987a] attractive. Instead of using cryptography, their *simulated broadcast primitive* relies upon the idea of letting all processes witness a process's message broadcast. As mentioned above, one could combine the reliable channel implementation of Section 5.1 with their primitive, leading to each broadcast taking four rounds. Instead we present an alternative primitive which achieves the properties of authenticated channels in three rounds.

The definition of the authenticated reliable broadcast of message m by process s in round k is based on (i) the broadcast primitive *authenticated-broadcast*(m) and (ii) the reception callback *authenticated-accept*(s, m, k). The semantics of the broadcast primitive are fully captured by the following three uniform properties [Hadzilacos and Toueg 1993]:

Correctness: If a non-faulty process p executes *authenticated-broadcast*(m) in round k,

```
1:  variables
2:      r ∈ ℕ                                              // the current macro round number
3:      witness(s, m, k)                  // the set of processes who witnessed s sending m in round k

4:  procedure authenticated-broadcast(m)
5:      send(INIT, r, m) to all nodes [including itself]

6:  thread authenticated-receive :
7:      in each micro-round:
8:      if received(INIT, k, m) from s in round k then
9:          send(ECHO, k, s, p) to all nodes [including itself]
10:     if received(ECHO, k, m, s, q) from q in some round ≥ k then
11:         send(confirm, k, m, s, q) to all nodes [including itself]
12:     if received(confirm, k, m, s, q) from at least n − t − 2ℓ distinct processes in some round ≥ k then
13:         witness(s, m, k) = witness(k, m, s) ∪ {q}
14:     if |witness(s, m, k)| ≥ n − t − ℓ then
15:         authenticated-accept(s, m, k)
```

Algorithm 5.2. *Broadcasting primitive for simulated authentication, code for process* p

then every correct process executes *authenticated-accept*(p, m, k) in the same round.

Uniform Unforgeability: If a correct process p never executes *authenticated-broadcast*(m), then no correct process ever executes *authenticated-accept*(p, m, k).

Uniform Relay If a correct process executes *authenticated-accept*(p, m, k) in round $r \geq k$, then every correct process also executes *authenticated-accept*(p, m, k) in round $r + 1$ or earlier.

Note that the late delivery of the relay property can (by definition) only occur, if the initial broadcaster p is faulty. In fact, the collusion of a faulty broadcaster with faulty witnesses could defer the delivery of a message (through *authenticated-accept*) arbitrarily long;, it could even prohibit termination of the broadcast (this was also noted by Srikanth and Toueg [1987a] for their broadcasting primitive). However, this can only happen if no correct process has accepted the message in question yet, as otherwise, the relay property would ensure that all correct processes accept the message by the end of the round following acceptance by the first non-faulty process.

One particularly important detail of Algorithm 5.2 is that it re-sends all messages in every round and only uses messages received in the current round in its rules. This is necessary, because link failures can produce ℓ^a erroneous (echo, p, m, k) messages per round at any correct process, which could otherwise accumulate over multiple rounds to surpass any threshold.

In our first lemma, we prove what can be seen as variants of the Correctness and Unforgeability properties, which are concerned with the some process p sending of an ECHO message and p being added to the set *witness* at all correct processes.

Lemma 5.2. *When (i) a correct process* p *sends an* ECHO *message (for some broadcast) every correct process adds* p *to its set of witnesses at most two micro-rounds later. When (ii) a correct process* p *never sends an* ECHO *message for some broadcast then no correct process ever adds* p *to its witness set for that broadcast, provided that* $n > 2t + 2\ell + 2\ell^a$.

Proof. (i) When the correct process p sends an ECHO for a message, all but ℓ correct processes will receive it and respond with a *confirm* message. Every correct process thus

receives $n - t - 2\ell$ confirmations for p's ECHO message, and thus add p to their *witness* set for the message.
(ii) When process p does not send an ECHO for some broadcast, at most $t + \ell^a$ will "respond" with sending a *confirm* message, which allows for $t + 2\ell^a$ such messages to be received at any correct process. This, however, does not suffice to cause any correct process to add p to its *witness* set, or perform any other action, that might cause this indirectly (because $t + 2\ell^a < n - t - 2\ell$). □

Theorem 5.3 (Simulated Broadcast)**.** *Correctness, Unforgeability and Relay hold with* 3 *micro-rounds per macro-round assuming* $n > 2t + \ell + \ell^a + max(t, \ell + \ell^a)$.

Proof. Correctness: When a correct process s sends INIT, then all correct processes except ℓ receive that message. All those processes respond with an ECHO message tagged with their own identifier. By Lemma 5.2 they get all added to the *witness* set at each correct process. These sets thus have at least $n - t - \ell$ members which causes the message sent by s to be delivered within 3 micro-rounds, or one macro-round.
Unforgeability: If s does not send an INIT message for some message m in round r, then no correct process (except ℓ^a) will send a matching ECHO message. That is, at most $t + \ell^a$ processes send ECHO messages. From (ii) in Lemma 5.2 it follows that at most ℓ^a correct processes will be added to the *witness* set for this message. Therefore, the maximum size of such a set is bounded by $t + \ell^a$, which in turn implies that the message cannot be delivered, since $t + \ell^a < n - t - \ell$.
Relay: When some (correct) process delivers s, m, k then at least $n - t - \ell$ processes must be in its *witness* set for this message. From (ii) in Lemma 5.2, it follows that all correct processes (at least $n - t - \ell - t$) in the *witness* set must have sent an ECHO message. Therefore, by (i) in Lemma 5.2 every correct process has at least $n - t - \ell - t > t + \ell^a$ processes in its *witness* set. This implies that every correct process will send an ECHO message. When all correct processes send ECHO messages, every correct process will add every correct process into its *witness* set, and thus this set will surely grow to $n - t > n - t - \ell$ processes, resulting in the delivery of the message at most two micro-rounds later. Since these two micro-rounds may span over a macro-round change, the delivery may occur only in the next macro-round.

By combining $n - t - \ell - t > t + \ell^a$, (i.e., $n > 3t + \ell + \ell^a$) as required for Relay (and Unforgeability) and $n > 2t + 2\ell + 2\ell^a$ as required for Lemma 5.2, one obtains a bound of $n > 2t + \ell + \ell^a + max(t, \ell + \ell^a)$. □

By plugging in the hybrid simulated broadcast primitive of Algorithm 5.2 into an Byzantine agreement algorithm assuming authentication one gets a non-authenticated algorithm for Byzantine agreement that tolerates Byzantine process and both arbitrary faulty and lossy links. In particular, one could choose the algorithm used by Srikanth and Toueg [1987a] (or for that matter the original one by [Dolev and Strong 1982b]). Since these algorithms require $n > t$ only, the composed protocol "inherits" the resilience bound $n > 2t + \ell + \ell^a + max(t, \ell + \ell^a)$ from our broadcasting primitive. Moreover, the two algorithms can even be simplified: the algorithms explicitly forward all messages they receive after adding their signature. This is in fact not necessary, due to the relay property, which performs this forwarding implicitly.

CHAPTER 6

$\mathcal{A}_{T,E}$ AND HER IMPLEMENTATION

Although originally considered only for synchronous systems round models can provide useful abstractions for capturing the synchrony in general systems. (This is exemplified by the results of Chapter 3.) In the context of not fully synchronous systems one can separate round models into two classes depending on what happens with messages that did not arrive within a round. As already mentioned the model presented in Section 3.2.4 is an example of a communication open model as late messages are simply delivered whenever they arrive in later rounds. Informally one could say that the processes are always open to receive late information. The alternative are communication closed models [Elrad and Francez 1982] in which messages are never delivered after the round they were sent in has ended. That is, late messages are simply discarded. This approach has the advantage that not only synchrony but also reliability of communication can be captured in the same way, as there is no difference between a late, a lost, and a message never sent. This is the idea behind the Heard-Of Model [Charron-Bost and Schiper 2006a; Charron-Bost and Schiper 2009], which provides a framework for modelling different system assumptions based on predicates on $HO(p, r)$ sets, that is, the set of processes p has *heard of* in round r model. As the reasons for some process q not being in $HO(p, r)$ are completely abstracted away, q cannot be the culprit. In other words, there are no faulty processes in the *HO*-model.

We show that this approach can be generalized for systems with non-benign failures [Biely et al. 2007a]. Typically a process cannot distinguish whether a faulty message originates from a communication fault and from to a process fault. Therefore, from an algorithm designer's point of view the important information is how many of the messages received in a round can be trusted. This can be captured by the safe heard-of set $SHO(p, r)$ which comprises those processes p has correctly received information from in round r. Clearly, unlike the *HO* set this set is not known to the process at run-time. The algorithm can only rely on the assumptions made about the size of this set.

So while the heard-of sets capture liveness of communication, one can regard the safe heard-of sets to capture communication safety. These collections of sets allow us to specify sufficient conditions for solving consensus in such a system. We will deter-

mine predicates sufficient for solving consensus with a certain algorithm in Section 6.2. We will then look into a lower level model (based on a generalization of the partially synchronous model [Dwork et al. 1988]) and try to implement these predicates. Unfortunately, due to formal reasons and the particularities of the model we will have to give up the clear separation between the round model implementation and the consensus algorithm. Informally, the problem is that it is not possible to guarantee that a process is not "in the same round twice" in our model.

6.1. Heard-Of Sets and Communication Predicates

Computations in our model are structured in rounds, which are communication-closed layers. Algorithms in the round model consist of the following components (for each process p): the set of possible states denoted by $states_p$, a subset $init_p$ of initial states, and for each positive integer r called *round number*, a message-sending function S_p^r mapping $states_p \times \Pi$ to a unique message from $\mathcal{M}$, and a state-transition function T_p^r mapping $states_p$ and partial vectors (indexed by Π) of elements of $\mathcal{M}$ to $states_p$. The collection of algorithms for the processes is called an *algorithm on* Π.

Each process p progresses through the computation by iterating through rounds $r \in \mathbb{N}$. In each round r, a process p will

1. apply S_p^r to the current state s_p^r, and emit the "messages" to be sent (according to its sending function S_p^r) to each process;
2. then determine the partial vector $\vec{\mu}_p^r$, formed by the messages that p receives in round r;
3. apply T_p^r to its current state and $\vec{\mu}_p^r$, which yields s_p^{r+1}.

Mapping this behaviour to the generic model of Chapter 2 is pretty straightforward: We only have to define how the above course of action is mapped to steps. The simplest assumption is to assume that processes perform all three items in one step. However, as messages are in transit between the first two items, we assume that one step consists of receiving the messages of round r, then applying T_p^r, and finally sending the messages determined by S_p^{r+1}. Recall that in the *HO*-model there is not notion of faulty processes, which nicely matches our assumption in the generic model that processes always follow their algorithm. As *HO*-algorithms are based solely on the sets of received messages (or rather guarantees about them), we abstain from giving the reasons for missing or corrupted messages. In other words, we do not provide detailed failure assumptions at this level of abstraction. (We will do so in Section 6.3, however.)

The partial vector $\vec{\mu}_p^r$ is called the *reception vector of* p *at round* r. The state s_p^r a process is in at the beginning of round r is subsequently also referred to as p's round r state. For any part of this state, that is any variable x_p local to process p, we denote $x_p^{(r)}$ the value of x_p in s_p^r, i.e., at the beginning of round r.

Each *run* is entirely determined by the initial configuration (i.e., the collection of process initial states), and the collection of the reception vectors $\left(\vec{\mu}_p^r\right)_{p\in\Pi,\ r>0}$. For each process p and each round r, we introduce two subsets of Π:

- The *heard-of* set, denoted $HO(p, r)$, which is the support of $\vec{\mu}_p^r$, i.e.,

$$HO(p, r) = \left\{ q \in \Pi : \vec{\mu}_p^r[q] \text{ is defined} \right\}.$$

- The *safe heard-of* set, denoted $SHO(p, r)$, and defined by

$$SHO(p, r) = \left\{ q \in \Pi : \vec{\mu}_p^r[q] = S_q^r(s_q, p) \right\}.$$

Both sets specify the difference between *what should be sent* and *what is actually received*. As for the benign case [Charron-Bost and Schiper 2009], we make no assumption on the reason why $\vec{\mu}_p^r[q] \neq S_q^r(p, s_q)$: it may be due to an incorrect sending by q, an incorrect receipt by p, or due to the corruption by the link from q to p. Obviously, we have $SHO(p, r) \subseteq HO(p, r)$. In contrast to $HO(p, r)$, process p is not able to determine $SHO(p, r)$.

From the two sets $HO(p, r)$ and $SHO(p, r)$, we form the *altered heard-of set* denoted $AHO(p, r)$ as follows:

$$AHO(p, r) = HO(p, r) \setminus SHO(p, r).$$

For any round r we can define the *altered span* denoting the set of processes from which at least one process received a corrupted message, and we can then extend this definition to whole executions:

$$AS(r) = \bigcup_{p \in \Pi} AHO(p, r) \qquad\qquad AS = \bigcup_{r>0} AS(r)$$

6.1.1. HO Machines

A *heard-of machine* for a set of processes Π is a pair $\langle \mathcal{A}, \mathcal{P} \rangle$, where $\mathcal{A}$ is an algorithm on Π, and $\mathcal{P}$ is a *communication predicate*, i.e., a predicate over $(HO(p, r))_{p \in \Pi,\, r>0}$ and $(SHO(p, r))_{p \in \Pi,\, r>0}$.

As an example, we can model the classical assumption, that no more than α processes may send a corrupted information in a computation:[1]

$$\mathcal{P}_\alpha^{perm} := |AS| \leq \alpha \tag{6.1}$$

For our algorithms we will consider the weaker predicate $\mathcal{P}_\alpha$ that restricts the number of corrupted messages only per round and per process:

$$\mathcal{P}_\alpha := \forall r > 0, \forall p \in \Pi : |AHO(p, r)| \leq \alpha \tag{6.2}$$

with $\alpha \in \mathbb{R}$ such that $0 \leq \alpha \leq n$, and we say that a computation has α-*safe communications* when $\mathcal{P}_\alpha$ holds. Note that $\mathcal{P}_\alpha^{perm}$ implies $\mathcal{P}_\alpha$.

We will consider only communication predicates, that are *time-invariant*: A communication predicate $\mathcal{P}$ is time-invariant if it has the same truth value for all heard-of collections $(HO(p, r + i))_{p \in \Pi, r>0}$ and $(SHO(p, r + i))_{p \in \Pi, r>0}$ for any $i \in \mathbb{N}$. This allows processes to start in their execution in any round r, while preserving the truth value of $\mathcal{P}$. The predicates $\mathcal{P}_\alpha^{perm}$ and $\mathcal{P}_\alpha$ are trivially time-invariant, since they are *permanent* predicates. Time-invariant predicates that characterize *eventual properties* typically have the form

$$\forall r > 0 \; \exists r_0 > r : \; \mathcal{P}_{r_0},$$

[1] Note that in the classical "Byzantine" setting, also the state of a "faulty" process can be corrupted, which is not the case in our model. Refer to Chapter 7 for a discussion.

where $\mathcal{P}_{r_0}$ is a predicate over the collection $(HO(p,r); SHO(p,r))_{p\in\Pi}$. In other words, such properties always hold eventually, which contrasts the time-variant predicate $\exists r_0 > 0 : \mathcal{P}_{r_0}$ which may only holds once.

HO-machines for the benign case [Charron-Bost and Schiper 2009] can be seen as a special case of the above definition, with $AS = \emptyset$, which is equivalent to assuming the predicate

$$\mathcal{P}_{benign} := \forall p \in \Pi, \forall r > 0 : \ SHO(p,r) = HO(p,r).$$

6.1.2. Consensus

Since, contrary to classical approaches, there is no deviation according to T_p^r, that is, there is no notion of a faulty process, we have to adapt the formulation of the consensus problem such that it holds for all processes. Thus we get the following properties:

Validity: If all processes have the same initial value this is the only possible decision value.
Agreement: No two processes may decide differently.
Termination: All processes eventually decide.

Formally, an HO machine $\langle \mathcal{A}, \mathcal{P} \rangle$ solves consensus, if any run for which $\mathcal{P}$ holds, satisfies Validity, Agreement, and Termination. To make this definition non-trivial, we assume that the set of *HO* and *SHO* collections for which $\mathcal{P}$ holds is non-empty.

6.2. The $\mathcal{A}_{T,E}$ algorithm

In this section we present the $\mathcal{A}_{T,E}$ algorithm, and determine the predicates under which it is correct. The code of $\mathcal{A}_{T,E}$ is given as Algorithm 6.1.

In $\mathcal{A}_{T,E}$ each process p maintains a variable x_p initially equal to p's initial value. At each round, process p broadcasts x_p, and if it receives more than T ("Threshold") messages, updates x_p, by setting x_p to the smallest of the values it has received the most frequently. If p has received more than E ("Enough") times the same value v, then it decides on v.

Basically, it consists in a parametrization of the two "receive thresholds" T and E of the *OneThirdRule* algorithm [Charron-Bost and Schiper 2006a; Charron-Bost and Schiper 2009] for solving consensus in the presence of benign failures (there both are equal to $\frac{2n}{3}$). The main features of *OneThirdRule* are: (i) the algorithm is always safe[2], whatever the number of benign transmission faults is, and (ii) the algorithm is *fast* in the sense that it requires two rounds to terminate in every failure-free run, and further it terminates at the end of the first round if all initial values are equal. In the sequel, we show that, for appropriate choices of T and E, the $\mathcal{A}_{T,E}$ algorithm both tolerates a large number of benign faults (omission) and non-benign faults (corruption), while retaining the main features of *OneThirdRule*. Contrary to *OneThirdRule* $\mathcal{A}_{T,E}$ is not always safe, but requires a predicate limiting the amount of corrupted communication a process can receive (the predicate is called $\mathcal{P}_\alpha$ and is introduced above in Equation (6.2)). For deciding $\mathcal{A}_{T,E}$ requires a second predicate to hold as well, which is of the eventual flavour and referred to as $\mathcal{P}^{\mathcal{A},live}$ and given in Figure 6.1.

[2]By the algorithm being always safe, we mean that the safety properties of consensus (namely, Validity and Agreement) are guaranteed even in asynchronous runs.

1: **Variable Initialization:**
2: $x_p \in V$, initially v_p // v_p is the initial value of p

3: **Round** r**:**
4: S_p^r:
5: send $\langle x_p \rangle$ to all processes
6: T_p^r:
7: **if** $|HO(p,r)| \geq T$ **then**
8: $x_p :=$ the smallest most frequently received value
9: **if** at least E values received are equal to v **then**
10: DECIDE(v)

Algorithm 6.1. *The $\mathcal{A}_{T,E}$ algorithm*

$$\forall r_0 > 0,\ \exists r \geq r_0,\ \exists \Pi_r^1, \Pi_r^2 \subseteq \Pi \text{ s.t.} (\left|\Pi_r^1\right| \geq E - \alpha) \wedge (\left|\Pi_r^2\right| \geq T) \wedge (\forall p \in \Pi_r^1,\ HO(p,r) = SHO(p,r) = \Pi_r^2)$$
$$\wedge$$
$$\forall r > 0,\ \forall p \in \Pi,\ \exists r_p > r\ :\ |HO(p,r_p)| \geq T$$
$$\wedge$$
$$\forall r > 0,\ \forall p \in \Pi,\ \exists r_p > r\ :\ |SHO(p,r_p)| \geq E$$

Figure 6.1. *Predicate $\mathcal{P}^{\mathcal{A},live}$*

6.2.1. Correctness of the $\mathcal{A}_{T,E}$ algorithm

First we introduce some notation. For any value $v \in V$ and any process p, at any round $r > 0$, we define the sets $R_p^r(v)$ and $Q_p^r(v)$ as follows:

$$R_p^r(v) := \left\{ q \in \Pi :\ \vec{\mu}_p^r[q] = v \right\}$$
$$Q_p^r(v) := \left\{ q \in \Pi :\ S_q^r(p, s_q^r) = v \right\}.$$

In words, the set $R_p^r(v)$ represents the set of processes from which p receives v in round r, while $Q_p^r(v)$ is the set of processes that ought to send v to p in round r. Since every process sends the same message to all in $\mathcal{A}_{T,E}$, the sets $Q_p^r(v)$ do not depend on p, and so can be just denoted by $Q^r(v)$ without any ambiguity.

We start our correctness proof with a general basic lemma.

Lemma 6.1. *For any process p and any value v, at any round r, we have:*

$$\left|R_p^r(v)\right| \leq |Q^r(v)| + |AHO(p,r)|.$$

Proof. Suppose that process p receives a message with value v at round $r > 0$ from process q. Then, either the code of q prescribes it to send v to p at round r, i.e., q belongs to $Q^r(v)$ and thus q is also in $SHO(p,r)$, or the message has been corrupted and q is in $AHO(p,r)$. It follows that

$$R_p^r(v) \subseteq Q^r(v) \cup AHO(p,r),$$

which implies

$$\left|R_p^r(v)\right| \leq |Q^r(v)| + |AHO(p,r)|. \qquad \square$$

Lemma 6.1 naturally leads to consider the communication predicate $\mathcal{P}_\alpha$ from (6.2): $\mathcal{P}_\alpha$ bounds the size of $AHO(p,r)$, that is limits the discrepancy between the sets R_p^r and Q^r.

Our second lemma shows that choosing $E \geq \frac{n}{2}$ renders the decision rule in the $\mathcal{A}_{T,E}$ algorithm "deterministic".

Lemma 6.2. *If $E > \frac{n}{2}$, then the guard in line 9 of the $\mathcal{A}_{T,E}$ algorithm is true for at most one value v.*

Proof. Assume by contradiction that there exist a process p and a round r, so that the guard in line 9 is true for two distinct values v and v'. The code implies that $\left|R_p^r(v)\right| \geq E$ and $\left|R_p^r(v')\right| \geq E$. Since v and v' are different, $R_p^r(v)$ and $R_p^r(v')$ are disjoint sets, and so

$$\left|R_p^r(v) \cup R_p^r(v')\right| = \left|R_p^r(v)\right| + \left|R_p^r(v')\right|.$$

From $E > \frac{n}{2}$ we have $\left|R_p^{r_0}(v) \cup R_p^{r_0}(v')\right| > n$, a contradiction. □

As an intermediate step to argue agreement, our next lemma shows that a stronger condition on E ensures no two processes can decide differently *at the same round*:

Lemma 6.3. *If $E > \frac{n}{2} + \alpha$ then in any run of the HO machine $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \rangle$ there is at most one possible decision value per round.*

Proof. Assume by contradiction that there exist two processes p and q that decide on different values v and v' in some round $r > 0$. From the code of $\mathcal{A}_{T,E}$, we deduce that $\left|R_p^r(v)\right| \geq E$ and $\left|R_q^r(v')\right| \geq E$. Then Lemma 6.1 ensures that $|Q^r(v)| \geq E - \alpha$ and $|Q^r(v')| \geq E - \alpha$ when $\mathcal{P}_\alpha$ holds.

Since each process sends the same value to all at each round r, the sets $Q^r(v)$ and $Q^r(v')$ are disjoint if v and v' are distinct values . Hence $|Q^r(v) \cup Q^r(v')| = |Q^r(v)| + |Q^r(v')|$. Consequently, when $E > \frac{n}{2} + \alpha$, we derive that $|Q^r(v) \cup Q^r(v')| > n$, a contradiction. □

The next lemma ensures that for sufficient large T, once a process has decided, other processes may learn only the decision value *in that round*:

Lemma 6.4. *If $T > 2(n + 2\alpha - E)$, then in any run of the HO machine $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \rangle$ such that process p decides value v at round r_0, every process q that updates its variable x_q at round r_0 sets it to v.*

Proof. Suppose that process p decides value v at round $r_0 > 0$. Let q be any process such that $|HO(q, r_0)| \geq T$, i.e., q modifies x_q at round r_0. Let $Q^{r_0}(\bar{v})$ denote the set of processes that, according to their sending functions, ought to send messages different from v at round r_0, and let $R_q^{r_0}(\bar{v})$ denote the set of processes from which q receives values different from v at round r_0. Since each process sends a message to all at each round, $Q^{r_0}(\bar{v}) = \Pi \setminus Q^{r_0}(v)$, and thus $|Q^{r_0}(\bar{v})| = n - |Q^{r_0}(v)|$. Similarly, we have $R_q^{r_0}(\bar{v}) = HO(q, r_0) \setminus R_q^{r_0}(v)$, and since $R_q^{r_0}(v) \subseteq HO(q, r_0)$, it follows that $\left|R_q^{r_0}(\bar{v})\right| \leq T - R_q^{r_0}(v)$.

Since p makes a decision at round r_0, by line 7, $\left|R_p^{r_0}(v)\right| > E$. Then Lemma 6.1 implies $|Q^{r_0}(v)| > E - \alpha$, from which $|Q^{r_0}(\bar{v})| < n - (E - \alpha)$ follows. With an argument similar to the one used in the proof of Lemma 6.1, we derive that $\left|R_q^{r_0}(\bar{v})\right| \leq |Q^{r_0}(\bar{v})| + |AHO(q, r_0)|$. When $\mathcal{P}_\alpha$ holds, we obtain $\left|R_q^{r_0}(\bar{v})\right| \leq n + 2\alpha - E$.

From $\left|R_q^{r_0}(\bar{v})\right| \leq T - R_q^{r_0}(v)$ it follows that in order to ensure $\left|R_q^{r_0}(v)\right| > \left|R_q^{r_0}(\bar{v})\right|$ then one needs $T > 2(n + 2\alpha - E)$. This implies that v is the most frequent value received by q at round r_0. Then the code entails q to set x_q to v. □

We now extend the statement of Lemma 6.4 to hold also for any round after the decision:

Lemma 6.5. *If $T > 2(n + 2\alpha - E)$, then in any run of the HO machine $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \rangle$ such that process p decides some value v at some round $r_0 > 0$, every process q that updates its variable x_q at some round $r \geq r_0$ necessarily sets it to v.*

Proof. Assume process p decides value v at round $r_0 > 0$. Let q be some process that updates x_q at some round $r \geq r_0$. Assume for now that

$$|Q^r(v)| = \left|\left\{p' : x_{p'}^{(r-1)} = v\right\}\right| \geq E - \alpha, \tag{6.3}$$

then the same argument as in Lemma 6.4 applies, and so the code entails q to set x_q to v at round r. In order to show (6.3) we use induction on r:

Base case: $r = r_0$. Since p decides v at round r_0, we have $\left|R_p^{r_0}(v)\right| \geq E$. By Lemma 6.1, we get $|Q^{r_0}(v)| \geq E - \alpha$ when $\mathcal{P}_\alpha$ holds. Moreover we know (from the code of $\mathcal{A}_{T,E}$) that $Q^{r_0}(v) = \left\{p' \in \Pi : x_{p'}^{(r_0)} = v\right\}$, and so the base case follows.

Inductive step: $r > r_0$. We know that $\left|\left\{p' \in \Pi : x_{p'}^{(r-1)} = v\right\}\right| \geq E - \alpha$ from the inductive hypothesis. The same argument as in Lemma 6.4 gives $\left|\left\{p' \in \Pi : x_{p'}^{(r)} = v\right\}\right| \geq E - \alpha$, under the condition $T > 2(n + 2\alpha - E)$.

Thus the lemma follows. □

From the above lemmas, we derive a sufficient condition on E and T which enables the *HO* machine $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \rangle$ to satisfy the Agreement clause of consensus.

Proposition 6.6 (Agreement)**.** *If $E > \frac{n}{2} + \alpha$ and $T > 2(n + 2\alpha - E)$, then there is at most one possible decision value in any run of the HO machine $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \rangle$.*

Proof. Let $r_0 > 0$ be the first round at which some process p makes a decision, and let v be p's decision value. Assume that process q decides v' at round r. By definition of r_0, we have $r \geq r_0$.

We proceed by contradiction, and assume that $v \neq v'$. By Lemma 6.3, we derive that $r > r_0$. Since p decides v at round r_0 and q decides v' at round r, Lemma 6.1 ensures that $|Q^{r_0}(v)| \geq E - \alpha$ and $|Q^r(v')| \geq E - \alpha$ when $\mathcal{P}_\alpha$ holds. If $T > 2(n + 2\alpha - E)$, then Lemma 6.5 implies that $Q^{r_0}(v)$ and $Q^r(v')$ are disjoint sets. Therefore, $|Q^{r_0}(v) \cup Q^r(v')| = |Q^{r_0}(v)| + |Q^r(v')|$. Moreover, when $E > \frac{n}{2} + \alpha$, then it follows that $|Q^{r_0}(v) \cup Q^r(v')| > n$, which provides the contradiction. □

Similarly, we derive sufficient conditions on E and T which enable the *HO* machine $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \rangle$ to satisfy the Integrity clause of consensus.

Proposition 6.7 (Integrity)**.** *If $E > \alpha$ and $T > 2\alpha$ then in any run of the HO machine $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \rangle$ such that all the initial values are equal to some value v_0, then v_0 is the only possible decision value.*

Proof. Consider a run of the *HO* machine $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \rangle$ such that all the initial values are equal to v_0. First, we show by induction on r that:

$$\forall r > 0 : \; Q^r(v_0) = \Pi$$

Note that according to the $\mathcal{A}_{T,E}$'s code, p belongs to $Q^r(v_0)$ if and only if $x_p^{(r)} = v_0$, and so $Q^r(v_0) = \left\{p \in \Pi : x_p^{(r)} = v_0\right\}$.

Base case: $r = 1$. If all the initial values are equal to v_0, then every process sends a message with value v_0 at round 1.

Inductive step: Let $r > 1$, and suppose that $Q^{r-1}(v_0) = \Pi$. Let p be a process that updates its variable x_p at round $r-1$. Since $AHO(p, r-1) \leq \alpha$, each process p receives at most α values distinct from v_0 at round $r-1$. Therefore, either p does not modify x_p at the end of round r which remains equal to v_0, or p receives strictly more than T messages at round r, and thus strictly more than $T - \alpha$ messages with value v_0 and at most α values different from v_0. In the latter case, p sets x_p to v_0 since $T \geq 2\alpha$. This shows that definitely, $x_p^{(r)} = v_0$. Therefore, $Q^r(v_0) = \Pi$.

Let p be a process that makes a decision at some round $r_0 > 0$. We have just shown that $Q^{r_0}(v_0) = \Pi$. When $|AHO(p, r_0)| \leq \alpha$ holds, p receives at most α messages with value different to v_0. Since $E \geq \alpha$, the code entails p to decide v_0 at round r. □

For liveness, we introduce the time-invariant communication predicate $\mathcal{P}^{\mathcal{A},live}$, given in Figure 6.1, which (i) enforces x_q's to eventually be identical, and (ii) guarantees that each process then hears of sufficiently many processes to make a decision.

Proposition 6.8 (Termination)**.** *If* $n \geq E > \frac{n}{2} + \alpha$ *and* $n \geq T > 2(n + 2\alpha - E)$*, then any run of the HO machine* $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \wedge \mathcal{P}^{\mathcal{A},live} \rangle$ *satisfies the Termination clause of consensus.*

Proof. As $\mathcal{P}^{\mathcal{A},live}$ holds by assumption there is a round such that the first part of the conjunction holds. Let r be a round such that:

$$\exists \Pi_r^1, \Pi_r^2 \subseteq \Pi \text{ s.t. } (\left|\Pi_r^1\right| \geq E - \alpha) \wedge (\left|\Pi_r^2\right| \geq T) : \\ (\forall p \in \Pi_r^1,\ HO(p, r) = SHO(p, r) = \Pi_r^2).$$

We note, that in round r all processes $p \in \Pi_r^1$, will update x_p to the same value v in line 8, as they all receive the same values in round r. Since $|\Pi_1^r| \geq E - \alpha$, it follows that $\left|Q^{r+1}(v)\right| \geq E - \alpha$. If $T > 2(n + 2\alpha - E)$, a similar argument as the one used in Lemma 6.5 shows that every process q that updates x_q at round $r' > r$ definitely sets it to v. Moreover, (the second clause of) $\mathcal{P}^{\mathcal{A},live}$ implies $\forall q \in \Pi,\ \exists r_q > r\ :\ |HO(q, r_q)| \geq T$, that is, every process q in $\Pi \setminus \Pi_1^r$ updates x_q after round r. Let r' be some round after all these r_qs. Then we deduce that for each process $q \in \Pi$, we have $x_q^{(r')} = v$. Finally, since (due to the third clause of $\mathcal{P}^{\mathcal{A},live}$)

$$\forall p \in \Pi,\ \exists r_p > r'\ :\ |SHO(p, r_p)| \geq E,$$

we know that every process $p \in \Pi$ eventually receives strictly more than E messages with value v at some round $r_p > r'$, and so decides v. □

Combining Propositions 6.6, 6.7, and 6.8, we spin-off the following theorem:

Theorem 6.9. *If* $n \geq E > \frac{n}{2} + \alpha$ *and* $T > 2(n + 2\alpha - E)$*, then the HO machine* $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \wedge \mathcal{P}^{\mathcal{A},live} \rangle$ *solves consensus.*

Proof. The Agreement clause is a straightforward consequence of Proposition 6.6. For Integrity, we just check that $E > \alpha$ and $T > 2\alpha$ are both ensured by $n \geq E > \frac{n}{2} + \alpha$ and $T > 2(n + 2\alpha - E)$, respectively. Indeed for all $\alpha \geq 0$, we have:

$$E > \frac{n}{2} + \alpha \ \Rightarrow\ E > \alpha.$$

Moreover

$$E \leq n \ \wedge \ T > 2(n + 2\alpha - E) \ \Rightarrow \ T > 2\alpha.$$

Termination directly follows from Proposition 6.8. □

At this point we have to examine whether there are T and E such that $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \wedge \mathcal{P}^{\mathcal{A},live} \rangle$ solves consensus, for any integer α and a suitable n (clearly, $0 \leq \alpha \leq n$). Theorem 6.9 shows that it is sufficient to solve the following inequations in order to answer this question:

$$n \geq E > \frac{n}{2} + \alpha \tag{6.4}$$

$$n \geq T > 2(n + 2\alpha - E) \tag{6.5}$$

Setting $E = \frac{n}{2} + \alpha + \epsilon$ ($\epsilon > 0$) one gets, $n = 2\alpha + 2\epsilon$ from (6.4). Substituting this into the right side of (6.5), leads to $\alpha < \frac{n}{4}$.

Conversely, starting from $\alpha = \frac{n}{4} - \epsilon$ with $\frac{n}{4} \geq \epsilon > 0$ and using $E = T = n$, we get $n > \frac{3n}{4}$ from (6.4), we get $n > n - 4\epsilon$ from (6.5), so there exist values for T and E such that (6.4) and (6.5) can hold, and thus the *HO* machine $\langle \mathcal{A}_{T,E}, \mathcal{P}_\alpha \wedge \mathcal{P}^{\mathcal{A},live} \rangle$ can solve consensus.

This naturally leads us to question what are the "best choices" for T and E, for any given integer α, $0 \leq \alpha < \frac{n}{4}$. Roughly speaking, in $\mathcal{P}^{\mathcal{A},live}$, T plays the role of a minimal "liveness communication threshold" whereas E plays the role of a minimal "safety communication threshold" guaranteeing $\mathcal{A}_{T,E}$'s correctness. Thus, the best choices are for T and E as small as possible. However, T and E are not independent of each other because of (6.5). That is, the "best choices" for T and E are paradoxical, and so there is no best choice for T and E without specifying additional prerequisites on T and E.

Without such specific assumptions on communications (which could provide guidance as to how to find a suitable trade-off), we look for T and E such that $E = T$. In this case, (6.5) leads to $n \geq E = T > \frac{2}{3}(n + 2\alpha)$. The above discussions can be summarized as follows:

Proposition 6.10. *For any integer $0 \leq \alpha < \frac{n}{4}$, the HO machine $\langle \mathcal{A}_{E,E}, \mathcal{P}_\alpha \wedge \mathcal{P}^{\mathcal{A},live} \rangle$ with $E > \frac{2}{3}(n + 2\alpha)$ solves consensus.*

Note that we get $E = T > \frac{2n}{3}$ in the benign case (i.e., $\alpha = 0$), and that $\mathcal{A}_{\frac{2n}{3},\frac{2n}{3}}$ coincides[3] with the *OneThirdRule* algorithm [Charron-Bost and Schiper 2006a; Charron-Bost and Schiper 2009].

Interestingly Theorem 6.9 shows that with $\mathcal{A}_{T,E}$ we do not lose any of the basic properties of the *OneThirdRule* algorithm even in the presence of corrupted communications. Like *OneThirdRule*, $\mathcal{A}_{T,E}$ is quite resilient since it just requires $\mathcal{P}_\alpha$ to hold in order to be safe. Furthermore, from each initial configuration, there is at least one run of $\mathcal{A}_{T,E}$ that achieves consensus in two rounds, and in the case where all the initial values are equal, there is a run that achieves consensus just in one round. As the *OneThirdRule* algorithm, $\mathcal{A}_{T,E}$ is thus *fast* in the sense of [Lamport 2005].

[3] To be exact, it only coincides when we change $\mathcal{A}_{T,E}$ by replacing "$\geq T$" and "at least E" in lines 7 and 8 of Algorithm 6.1, respectively, by "$> T$" and "more than E (thus arriving at the version of $\mathcal{A}_{T,E}$ as originally proposed by Biely et al. [2007a]). This, however, does not (substantially) change any of the proofs.

6.3. Implementing $\mathcal{A}_{T,E}$'s Predicates

As we have observed above, the HO-model is a rather abstract model. In this section we look into the problem of implementing the round model (with predicates that allow $\mathcal{A}_{T,E}$ to solve consensus) atop of a specific synchrony model, which is similar to our translation of algorithm from the ROUND model into more basic models in Chapter 3.

Like Hutle and Schiper [2007b] we are interested in providing an algorithm that ensures some set of HO model predicates atop of a lower level model with dynamic (i.e., transient) process failures, with the difference being that Hutle and Schiper [2007b] only considered benign failures, whereas we will allow processes to be Byzantine. That is, low level process failures will be mapped to their outgoing transmission being faulty. For simplicity we do not allow any transmission failures to occur on links between two correct processes. As we assume that process failures are dynamic (that is, a failure is assumed to be transient and not fixed to a process), the question arises "how dynamic" failures may be. On the one hand we must not allow dynamic failures only in some subset of the processes, as this could be seen as permanent failures with a very specific failure assumption. On the other hand it is quite obvious, that we cannot allow all processes to be faulty at the same time. This would render consensus impossible to be solved: the Validity property cannot be ensured in such cases, as processes may forget about their initial values while being faulty. The key to our failure model is to assume that all processes may be faulty during an execution, but not at the same time. Hutle and Schiper [2007b] have shown that, in the case of benign faults, it is possible to define such a restriction on the dynamic nature of failures, and that consensus can be solved even if all processes may crash and recover infinitely often — processes that do so are called unstable — as long as there is a time window where they are up long enough. Note that in this respect they partially contradict previous claims that consensus is impossible to solve when all processes may be unstable [Aguilera et al. 2000]. However, this impossibility was shown in a system with failure detectors, where there is no notion of "(up) long enough" — in fact, such notion cannot be expressed in asynchronous systems, even when equipped with failure detectors.

When considering recovery in the case of benign faults, one distinguishes between system with or without stable storage [Hutle and Schiper 2007b; Aguilera et al. 2000]. In our case, stable storage is of no help, as a process can corrupt it arbitrarily when it is faulty. This makes the problem of solving consensus a little more intricate, as we have to rely on the information distributed in the system (as it is the case for benign recovery without stable storage) to let a recovering process learn about previous decision values or initial values. Unfortunately, in $\mathcal{A}_{T,E}$ one cannot find any provisions for such a reintegration except for the value selection in line 8. Moreover, processes do forget what rounds they have been in when they become faulty, which makes it impossible to ensure that no process is in a round twice — a basic assumption of round-based models.[4] Therefore, from a formal point of view a clear separation between the algorithm and the implementation is not possible. Consequently, our resulting algorithm is given as a monolithic one. (One may thus consider this section as being somewhat mis-titled.)

[4] In the benign case one can use stable storage to ensure this property [Hutle and Schiper 2007b].

6.3.1. Model and Problem Definition

In order to define properly what the statements "up long enough" and "being faulty at the same time" mean we will now instantiate the generic model introduced in Chapter 2. Our model is a relaxation of the partially synchronous model of Dwork et al. [1988] where the system is not eventually synchronous forever, but it is allowed to alternate between good and bad periods [Hutle and Schiper 2007b; Cristian and Fetzer 1999]. In good periods the synchrony assumptions hold for correct processes and messages between them, whereas in bad periods there is no such assumption.

Regarding the details of step atomicity we refer the reader to Section 3.2.3. In short, only a single message to another process[5] can be sent in a send step; in a receive step an arbitrarily large set S of messages can be received. In addition (and contrasting the original model), processes are equipped with time oracles in the form of unsynchronized local clocks which they may query at any time. For simplicity, we ignore clock drift. While all proofs can be adapted to include clock drift, it clutters up the presentation and makes their understanding cumbersome. Further, also for simplicity, we assume that clocks are always correct. However, it can easily be seen that this is only necessary in good periods, in bad periods only monotonicity is necessary. Note, that we do not attempt to synchronize these clocks.

The function determining the set of faulty processes $\mathcal{F}(\tau)$ can be arbitrary, but we assume that:

$$\forall p \in \Pi, \forall \tau \in \mathbb{T} : p \in \mathcal{F}(\tau^-) \ \wedge \ p \in \mathcal{C}(\tau) \implies$$
$$state(p, \tau) = recoverystate_p,$$

where τ^- denotes the left side limit of τ. In other words, processes can recover from being faulty only by entering a special recovery state.

Recall that we generalized the notion of faulty and correct for an interval $\mathcal{I} = [\tau_1, \tau_2) \subset \mathbb{T}$ as $\mathcal{F}(\mathcal{I}) = \bigcup_{\tau \in \mathcal{I}} \mathcal{F}(\tau)$ and $\mathcal{C}(\mathcal{I}) = \Pi \setminus \mathcal{F}(\mathcal{I}) = \bigcap_{\tau \in \mathcal{I}} \mathcal{C}(\tau)$. For an arbitrary contiguous time interval $\mathcal{I}$, we say a process p is $\mathcal{I}$-correct if it is not faulty at any time within that interval, i.e., $p \in \mathcal{C}(\mathcal{I})$; else it is $\mathcal{I}$-faulty, i.e., $p \in \mathcal{F}(I)$.

Before defining good and bad periods we introduce the following definitions:

For every contiguous interval $I \subset \mathbb{T}$ we say that the processes speed bound Φ holds in $\mathcal{I}$, if and only if every process p that is $\mathcal{I}$-correct takes at least one step (and only finitely many steps) in any contiguous subinterval of length Φ.[6]

For every contiguous interval $\mathcal{I} = [\tau_1, \tau_2) \subset \mathbb{T}$ we say that the communication bound Δ holds in $\mathcal{I}$, if and only if for any message sent from I-correct process p to I-correct process q at time τ_s, the following holds:

(i) If $\tau_s \in \mathcal{I}$, q makes a receive step at time $\tau_r \geq \tau_s + \Delta$ and $\tau_r \in \mathcal{I}$, then this message is received in this receive step or any receive step q made since τ_s

(ii) If $\tau_s < \tau_1$, q makes a receive step at time $\tau_r \geq \tau_1 + \Delta$ and $\tau_r \in \mathcal{I}$, then this message is received in this receive step or any receive step q made since τ_s or it is never received.

[5] That is, sending a message to oneself does not require a send step.

[6] There is a subtle difference to the model of Dwork et al. [1988] here. We use continuous time and not discrete time so that our definition of Φ does not put an upper bound on the speed of Byzantine processes (as it is the case in [Dwork et al. 1988]). We think that this weaker modelling fits better the intuition of an arbitrary fault.

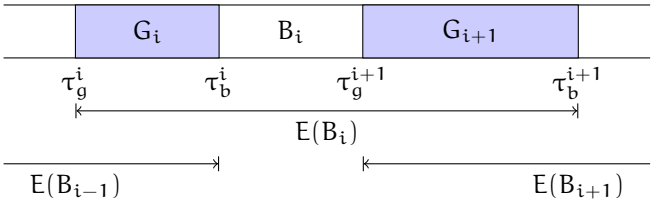


Figure 6.2. *Good, bad, and enclosing periods: definitions*

Note, that these definitions allow the actual reception by the algorithm to occur later than $\tau_s + \Delta$, which is necessary under certain conditions, for instance when the receiving process performed only send steps since τ_s. Moreover, our definition implies that a message from a bad period is received at most Δ after the start of a good period, or is lost.

We call an interval $\mathcal{I}$ a good period, if the process speed bound Φ and the communication bound Δ hold in $\mathcal{I}$. Our system alternates between good and bad periods, and we consider only maximal good periods of length $|\mathcal{I}| \geq W$, where the parameter W is to be determined later. We denote τ_g^i the beginning of the i^{th} good period with $|\mathcal{I}| \geq W$, and τ_b^i its end; $G_i = \left[\tau_g^i, \tau_b^i\right)$. Thus, $B_i = \left[\tau_b^i, \tau_g^{i+1}\right)$ denotes a bad period (cf. to Figure 6.2). From the above definitions, it follows that the system is in a bad period from $-\infty$ to at least τ_0, where we assume the computation to start by the first process making a step. This bad period is denoted B_0. For simplicity, and w.l.o.g. we assume $\tau_0 = 0$. Note that time 0 need not be the start of a good period.

We define $f(\mathcal{I}) = |\mathcal{F}(\mathcal{I})|$. If B_i (with $i > 0$) is a bad period, let $E(B_i)$ denote the *enclosing* period, i.e., the bad period with its two adjacent good periods; $E(B_0)$ is defined as the union of B_0 and G_1. In the remainder of this chapter we assume that the resilience bound t holds, that is for any bad period B_i, $f(E(B_i)) \leq t$. Moreover, we assume (for termination) that eventually there is a good period G_i with $|G_i| > D > W$ and $f(G_i) = 0$. Such good period is called deciding period.

In contrast to systems with perpetual faults, our definition of consensus allows processes to decide several times, but we guarantee that all decisions of correct processes are the same. Each process p has an initial value v_p taken from the set of all possible initial values V, and decides according to the following rules:

Agreement: When two processes p and q decide v_p and v_q at times τ_p and τ_q and $p \in \mathcal{C}(\tau_p) \wedge q \in \mathcal{C}(\tau_q)$, then $v_p = v_q$.
Validity: If all initial values are v and if p decides v_p at time τ_p and $p \in \mathcal{C}(\tau_p)$ then $v_p = v$.
Termination: Every process eventually decides.

6.3.2. Algorithm

Our algorithm (Algorithm 6.2) requires $n > 5t$, and uses the timeout parameters ϑ, η, and ζ. Determining the range of values for these parameters and the minimal length of good and deciding periods is carried out in the next subsection. Table 6.1 summarizes our results, where the "Typical values" listed in the table match the lower bound but are rounded to the next higher multiple of $n\Phi$.

The intuition behind our algorithm is the following: We use $\mathcal{A}_{T,E}$ (marked in red) as its core that reaches agreement on the input values. As we know from Section 6.2,

	Name	Lower Bound	Typical value	Upper bound
Retransmission delay	η	$n\Phi$		ϑ
ROUND message timeout	ϑ	$(3n-1)\Phi + 2\Delta$	$3n\Phi + 2\Delta$	
Maximum message age	ζ	$\vartheta + n\Phi + \Delta$	$4n\Phi + 3\Delta$	$W - (2n\Phi + \Delta)$
Minimum length of good period	W	$3\vartheta + (7n-4)\Phi + 3\Delta$	$16n\Phi + 9\Delta$	
Minimum length for decision	D	$W + \vartheta + (3n-1)\Phi + \Delta$	$22n\Phi + 12\Delta$	

Table 6.1. *Summary of Parameters*

$\mathcal{A}_{T,E}$ is based on a communication closed round structure, the remainder of the algorithm provides this abstraction. In order to do so it synchronizes rounds.

During a bad period rounds are in general not synchronized and thus there is no guarantee to receive all round r messages by the end of round r. However, the algorithm ensures that a group of correct processes keep their rounds synchronized and that no correct process is in a higher round than any member of this group. As soon as a sufficiently long good period starts, our algorithm ensures that all correct processes resynchronize with this group of processes and that all processes that have recovered update their state so that they are properly reintegrated. This ensures that after some bounded time in a good period, every correct process receives a message from every correct process in every round.

Furthermore, in order to avoid that information from a faulty process "spills over" to a period where the process is correct we have to ensure an "isolation" property, which is done by filtering the messages our algorithm stores. This and the other important concepts behind our solution are explained in more detail below.

Variables & Initialization. For a process p the estimate of the consensus decision is stored in x_p, the decision value in $decision_p$. When a process p normally starts its execution (by proposing a value v_p in line 1) of our algorithm it sets x_p to this input value. The algorithm starts in round 1, with r_p denoting p's current round number.

The information obtained from received messages is stored in three variables: For each round r', estimates from other processes are stored in $Rcv_p[r]$, where only the value for round the current round (r) and the following one ($r+1$) are kept. This variable is used in the consensus core of the algorithm, while the variables RF_p and FF_p are used in the round synchronization part. The latter contain the information if and when a ROUND resp. FIN message was received by p. In this context we use the following abbreviations in the algorithm with $\langle p, c\rangle$ denoting a pair of a process identifier and a clock-time:

$$Proc(F) := \{p \in \Pi \ : \ \exists c : \ \langle p, c\rangle, \in F\}$$
$$\#(v) := |\{q \ : \ Rcv_p[r_p][q] = v\}| .$$

Thus $Proc(F)$ denotes the set of processes which have an entry (with some time) in S. The abbreviation $\#(v)$ just counts the number of times v has been received in the current round, i.e., the number of entries in $Rcv_p[r_p]$ equal to v.

Progress in good periods. The aforementioned ROUND message is used for round synchronization as well — it indicates that the sending process has started a new round r (line 13). Subsequently the process sets a timeout of length ϑ and starts collecting messages. Once the timeout expires, it is ready to proceed to the next round, since it can be sure that in a good period all correct processes have received a ROUND message from all correct processes by that time. The process indicates this fact to the other processes by

1: **Propose** v_p
2: $x_p \leftarrow v_p$ // initial value
3: $r_p \leftarrow 1$
4: goto 9
5: **Recover**
6: $x_p \leftarrow \perp$ // or any other value $\notin V$
7: $r_p \leftarrow 0$
8: **Variable Initialization:**
9: $decision_p \leftarrow \perp$
10: $\forall r', q : Rcv_p[r'][q] \leftarrow \perp$
11: $\forall r' : RF_p[r'] \leftarrow \emptyset, FF_p[r'] \leftarrow \emptyset$

12: **while** true **do**
13: send $\langle \text{ROUND}, r_p, x_p \rangle$ to all
14: $timeout \leftarrow clock_p + \vartheta$
15: $r_{next} \leftarrow r_p$
16: **while** $clock_p \leq timeout$ **and** $r_p = r_{next}$ **do**
17: receive-and-process()
18: $r_{next} \leftarrow max\left\{r' : \left|\bigcup_{r'' \geq r'} Proc(RF_p[r''] \cup FF_p[r''])\right| \geq t+1\right\}$
19: $timeout \leftarrow -\infty$
20: **while** $r_p = r_{next}$ **do**
21: **if** $clock_p \geq timeout$ **then**
22: send $\langle \text{FIN}, r_p, x_p \rangle$ to all
23: $timeout \leftarrow clock_p + \eta$
24: receive-and-process()
25: **if** $|Proc(FF_p[r])| \geq n - t$ **then**
26: $r_{next} \leftarrow r_p + 1$
27: $r_{next} \leftarrow max\left(\{r_{next}\} \cup \left\{r' : \left|\bigcup_{r'' \geq r'} Proc(RF_p[r''] \cup FF_p[r''])\right| \geq t+1\right\}\right)$
28: **while** $r_p < r_{next}$ **do**
29: **if** $|\{q \in \Pi : Rcv_p[r_p][q] \neq \perp\}| \geq n - t$ **then**
30: $x_p \leftarrow \min\{v' : \#(v') = max_{v''}\{\#(v'')\}\}$
31: **if** $\exists v \in V : \forall q \in \Pi, Rcv_p[r_p][q] = v$ **then**
32: $decision_p \leftarrow v$
33: $r_p \leftarrow r_p + 1$

34: **procedure** receive-and-process()
35: $S \leftarrow$ receive step
36: **for each** $\langle q, \langle \text{ROUND}, r', \_\_ \rangle \rangle \in S$ **do**
37: $RF_p[r'] \leftarrow RF_p[r'] \cup \{\langle q, clock_p \rangle\}$
38: **for each** $\langle q, \langle \text{FIN}, r', \_\_ \rangle \rangle \in S$ **do**
39: $FF_p[r'] \leftarrow FF_p[r'] \cup \{\langle q, clock_p \rangle\}$
40: **for each** r' **do**
41: **for each** $\langle q, c \rangle \in RF_p[r']$ where $c < clock_p - \zeta$ **do**
42: remove $\langle q, c \rangle$ from $RF_p[r']$
43: **for each** $\langle q, c \rangle \in FF_p[r']$ where $c < clock_p - \zeta$ **do**
44: remove $\langle q, c \rangle$ from $FF[r']$
45: **for each** $\langle q, \langle \text{ROUND}, r', x' \rangle \rangle \in S$ and each $\langle q, \langle \text{FIN}, r', x' \rangle \rangle \in S$ **do**
46: **if** $r' \in \{r_p, r_p + 1\}$ **then**
47: $Rcv_p[r'][q] \leftarrow x'$

Algorithm 6.2. *Consensus in the Byzantine/recovery model, code for a process* p*, started by* **Propose** $v_p \in V$ *or* **Recover**

sending a FIN message. We say that a process is in the FIN phase of a round, when it has sent at least one of these message. In good periods this message serves only one purpose: it keeps a group of correct processes in a bad period "together", i.e., their round numbers differ by at most 1. Once a process has received $n - t$ FIN messages, it ends the round and advances the next round (line 26).

However, since good periods do not always start at the beginning of a round, FIN messages are retransmitted every η time (lines 21–23). This is in contrast to ROUND messages, which are sent only once per round. However, as FIN messages contain the same information as ROUND messages, they also serve as retransmissions of the ROUND messages. This allows the progression through the rounds to continue (shortly) after the beginning of a good period even if all messages are lost in a bad period. Note that in order to guarantee that processes will perform at least one reception step in between these retransmissions we have to assume that $\eta \geq n\Phi$.

Resynchronization. Not all correct processes will follow the group with the highest round numbers in a bad period. In order to resynchronize, the algorithm has a "catch-up" rule that allows such processes to resynchronize with this group. Once a process receives at least $t + 1$ messages with a round number r'' that is higher or equal to some higher round number $r' > r$, the algorithm immediately advances to round r' (lines 18 and 27). As there are at most t faulty processes in an enclosing period $E(B)$, faulty processes alone can not cause a premature end of a round for a correct process.

Note that this resynchronization also allows recovering processes (which re-start in round 0) to be reintegrated.

Message filtering. In order to deal with the additional challenges of the recovery model, we need to implement the aforementioned "isolation" property. This is achieved by the following two provisions (implemented by the function receive-and-process() in Algorithm 6.2):

(i) We record quorums for each round together with a time-stamp and remove entries that are older than ζ time. With the right choice of ζ, due to the "overlapping" definition of t it can be ensured that no message from a faulty process sent in a bad period survives a good period. Thus a changing set of faulty processes cannot harm the round structure.

(ii) We record in Rcv the values received through ROUND and FIN messages only for the current and the next round. Since (as we show in the next section) for two different bad periods, correct processes have distinct round numbers (actually we guarantee that there is one round in between), this prevents bad processes to place erroneous estimates into Rcv for rounds where they might be correct.

Both adaptions are thus designed to limit the impact of a Byzantine process of a bad period on later periods, when it is no longer faulty. In more detail, (i) prevents a Byzantine process to work together with the Byzantine processes of later bad periods in order to terminate rounds prematurely. By (ii) and the fact that in a good period the algorithm advances at least two rounds we ensure that no more than t messages of any round are corrupted.

Consensus core. As mentioned before the $\mathcal{A}_{T,E}$ part of our solution is marked in red in Algorithm 6.1. The sending part of a round S_p^r (line 4 of Algorithm 6.1) can be found in the sending of ROUND messages to all other processes at the beginning of a round

(line 13). Beside the current estimate x_p this message also contains the current round number. At the end of each round lines 29–32 are executed, which corresponds to calling the transition function of the $\mathcal{A}_{T,E}$ algorithm (cf. lines 7 to 10 in Algorithm 6.1). Here, we use the values stored in $Rcv_p[r_p]$ to determine the messages received in the current round.

Regarding the parameters of $\mathcal{A}_{T,E}$, it is clear that the model implies $\alpha \geq t$. Moreover, due to the message filtering we can set $\alpha = t$. In order to allow recovering processes to learn new values, we need to fix $T = n - t$, otherwise the faulty processes could prevent a process p that just has recovered from learning a suitable estimate x_p, by just not sending any message. From this it follows that $n > 5t$ and, in the minimum case $n = 5t + 1$, that $E = n$.

Recovery. When a process recovers it cannot start its execution as it would normally do, since it does not know what value to propose. Instead it starts its execution from line 5, where it uses a default value which is not one of the normally proposed values as its estimate. In a good period the recovering process is guaranteed to receive more than $n - t$ messages for a round, and will thus set its estimate to legal value, i.e., $x_p \in V$. Moreover, a recovering process p starts in round $r_p = 0$ instead of 1, which ensures that it never interfere with the round progress of the other processes.

6.3.3. Proofs

As a first step to prove the correctness of Algorithm 6.2 we start by showing that messages from a bad period have no influence on later bad periods. This is the key point that allows us to tolerate a changing set of Byzantine processes. In other words these lemmas show that our function receive-and-process() (lines 34 ff.) performs meaningful message filtering.

Lemma 6.11. *If $W > \Delta + 2n\Phi + \zeta$, then at time $\tau_g^i + \Delta + 2n\Phi + \zeta$ the sets RF_p and FF_p of any G_i-correct process p do no more contain entries from processes in $\mathcal{F}(E(B_{i-1})) \setminus \mathcal{F}(G_i)$ that were sent while the sender process was faulty.*

Proof. Since p is G_i-correct, any message m sent before τ_g^i arrives at p by time $\tau_g^i + \Delta$, and will thus be received during the first receive step following this time.[7] In our algorithm, processes have to take at least one receive step between two broadcasts, where the latter comprises at most $n-1$ send steps. Since $E(B_{i-1})$-correct processes make a step in a good period at least every Φ time units, m is received by p at the latest at time $\tau_g^i + \Delta + n\Phi$. After this time, RF_p and FF_p will no more be updated by messages from processes which are B_{i-1}-faulty but G_i-correct, and sent before τ_g^i. Finally, after ζ additional time, all entries received before $\tau_g^i + \Delta + n\Phi$ will be removed in the first reception step following $\tau_g^i + \Delta + n\Phi + \zeta$. This reception step will happen at $\tau_g^i + \Delta + n\Phi + \zeta + n\Phi$ or earlier. □

Lemma 6.12. *If $W > \Delta + 2n\Phi + \zeta$, then for any time τ at each τ-correct process p, the number of entries in FF_p and RF_p corresponding to messages sent when the sender was faulty is bounded by t.*

Proof. It follows directly from Lemma 6.11 that during $B_i \cup G_{i+1}$, any faulty entry in RF_p and FF_p corresponds to a message from a $E(B_i)$-faulty process. Together with the bound on failures we get the lemma. □

[7] Alternatively, it is also possible that such message is lost. In this case the lemma holds trivially.

In the last two lemmas we have established a lower bound on the length of good periods, which prevents messages of faulty processes $\mathcal{F}(E(B_i)) \setminus \mathcal{F}(E(B_{i+1}))$ from contaminating the *RF* and *FF* sets in B_{i+1} and later. But we must avoid removing messages from correct processes in good periods too early. That is, we must choose a ζ that prevents messages from being removed before an updated message (i.e., a more recent message for the same round) is guaranteed to arrive.

Lemma 6.13. *If $\zeta > max\{\vartheta, \eta\} + n\Phi + \Delta$, then no message sent by a G_i-correct process p to a G_i-correct process q in G_i is removed before an updated message has arrived, or before the end of G_i.*

Proof. In order to prove this lemma, we have to determine the maximal time between two reception steps of q that return a message from p. Clearly the worst case for this time is when the first message arrives as early as possible, whereas the second message arrives as late as possible.

For the fastest case, let $\tau \in G_i$ denote the time when p starts the broadcast in which it sends the first message to q. Since there is no lower bound on the time for all (up to $n-1$) send steps, nor on the message transmission delay, τ remains to be the only lower bound on when the first message is received at q.

For the slowest case, we let τ' denote the starting time of the second broadcast. Then we know that the $(n-1)$-th send step takes place at the latest at $\tau' + (n-1)\Phi$ and the transmission delay is Δ at most. The second message will thus be received in the first reception step following that. Note that we do not care when the reception step in which the second message is received occurs, as long as the first message will not be removed before that time, which leads to the requirement $\zeta \geq (\tau' - \tau) + (n-1)\Phi + \Delta$.

In order to determine the maximum time between two subsequent broadcasts, we note that (i) at most $\vartheta + \Phi$ after the last ROUND message was sent for some round, p will start to broadcast the FIN message for that round, and will (ii) repeat to do so every $\eta + \Phi$. Thus, $\tau' - \tau \leq max\{\vartheta, \eta\} + \Phi$, which shows the lemma. □

The next lemma deals with the amount of synchronization between processes, even in bad periods. Here, we show that a group of correct processes always remains "close" together. To do so we will use the "filtering" properties of the sets *RF* and *FF* shown above.

Lemma 6.14. *Assume $W > 2(\vartheta + 2n\Phi + 2) + \eta$. For any time τ, let i be such that $\tau \in \mathcal{I} = B_i \cup G_{i+1}$. Let r denote the highest round number among all $\mathcal{I}$-correct processes at time τ. Then at least $n - t - f(E(B_i))$ $\mathcal{I}$-correct processes are either in round r or are in the* FIN *phase of round $r-1$.*

Proof. Assume by contradiction that there is a time τ that is in some interval $\mathcal{I} = B_i \cup G_{i+1}$, such that at τ there are less than $n - t - f(E(B_i))$ $\mathcal{I}$-correct processes in rounds r and $r-1$. We first note that because of Lemma 6.12, in $\mathcal{I}$ there are no messages an $E(B_i)$-correct process sent in a previous bad period anywhere in the system, i.e., neither in transit nor in the variables *RF* or *FF* of an $E(B_i)$-correct process. By the definition of r, at least one $\mathcal{I}$-correct process is in round r. Let p be the $\mathcal{I}$-correct process that first entered round r (if there are several, fix one of them). Then p received $n - t$ FIN messages from round $r-1$ (i.e., p increased r_p in line 26, since it cannot have entered round r with $t+1$ round r

messages through lines 18 or 27). Since $\mathcal{I} \subset E(B_i)$, at most $f(E(B_i))$ processes are $\mathcal{I}$-faulty, and thus at least $n - t - f(E(B_i))$ of the $n - t$ messages were sent by $\mathcal{I}$-correct processes. Therefore they were in round $r - 1$ before τ, and by definition of r, are either still in $r - 1$ or in round r. Contradiction. □

From Lemma 6.14 we continue our proofs by analyzing what happens at the start of a good period. In a good period, we require *all* processes that are correct in this period to synchronize, so that there is a round where every correct process receives a message (ROUND or FIN) from every correct process (cf. Lemma 6.19).

Lemma 6.15. *Let* $\tau = \tau_g^i + max\{\vartheta, \eta\} + (3n - 2)\Phi + \Delta$. *If all* G_i*-correct processes are in round* r *at* τ_g^i *and* $\tau \in G_i$, *then by* τ *all* G_i*-correct processes are in round* $r + 1$.

Proof. Any G_i-correct process p is either (i) amidst a broadcast at τ_g^i, or (ii) waiting for a timeout (either ϑ or η) to expire. In case (ii), p will start broadcasting FIN messages for round r by $\tau_g^i + max\{\vartheta, \eta\} + \Phi$ and the broadcast will take $(n-1)\Phi$ time. In case (i), let $k \geq 0$ be the number of send steps p has already performed before τ_g^i. Then p will perform $n - k - 1$ more send steps in the current broadcast, start the timeout (either ϑ or η), wait for that to expire, and start the next broadcast at most Φ time after the timeout expired. Adding up the times above, with the time needed to perform the second broadcast, we arrive at $\tau_g^i + (n - k - 1)\Phi + max\{\vartheta, \eta\} + \Phi + (n - 1)\Phi \leq \tau_g^i + max\{\vartheta, \eta\} + 2(n - 1)\Phi$ as an upper bound for p having sent a FIN message for round r to all processes. (Note, that if p was broadcasting FIN at τ_g^i, the bound reduces to $\tau_g^i + (n - 1)\Phi + max\{\vartheta, \eta\} + \Phi + k\Phi = \tau_g^i + max\{\vartheta, \eta\} + n\Phi$, by assuming that processes always perform the sends of a broadcast in the same order.)

At most, Δ after the broadcasts of FIN messages has finished at all processes, these messages arrive at all processes, and are received in the next reception step after that time. As a process may be currently broadcasting when the message arrives, and a broadcast takes up to $(n - 1)\Phi$, it may take until $\tau_g^i + max\{\vartheta, \eta\} + 2(n - 1)\Phi + \Delta + n\Phi$ for that reception step to occur. At this time, all G_i-correct processes will have $n - f(G_i) \geq n - t$ FIN messages for round r, and thus switch to round $r + 1$ by line 26. □

Lemma 6.16. *Let* $\tau = \tau_g^i + max\{\eta, \vartheta\} + (2n - 1)\Phi + \Delta$. *If some* G_i*-correct processes are in round* r, *with* r *being maximal at* τ_g^i, *and* $\tau \in G_i$ *then by* τ *all* G_i*-correct processes are at least in round* $r - 1$.

Proof. From Lemma 6.14 it follows that at least $n - t - f(E(B_i))$ processes are in round $r - 1$ or r at τ_g^i. Since $n - t - f(E(B_{i-1})) > 3t$ there are more than $t + 1$ processes which will send messages for either round r or round $r - 1$, such that all G_i-correct processes will eventually exceed the bounds of lines 18 or 27, and thus set their round to at least $r - 1$ as required.

What remains is to determine the latest time when this may happen. The messages in question are guaranteed[8] to have been sent by $\tau_g^i + max\{\vartheta, \eta\} + (n - 1)\Phi$. Since the receivers may (as above) be amidst a broadcast, the reception step (and thus the update of r_{next}) may take until $\tau_g^i + max\{\vartheta, \eta\} + (n-1)\Phi + \Delta + n\Phi$. As no process can be currently waiting for its ϑ timeout to expire in a round less than $r-1$, the processes will also update r_p. □

[8] Recall the note in the proof of Lemma 6.16.

Lemma 6.17 (Slowest Progress)**.** *If at some time $\sigma \in G_i$, the last G_i-correct processes enter round s (line 13), then by $\sigma' = max\left\{\sigma + \vartheta + (n-1)\Phi, \tau_g^i + \eta\right\} + 2n\Phi + \Delta$ all G_i-correct processes enter at least round $s+1$, if $\sigma' \in G_i$.*

Proof. Let p be one of the last (when there are multiple processes) or the last G_i-correct processes entering round s. After entering round s, p sends out its $\langle \text{ROUND}, s, x_p\rangle$ messages to all other processes. This takes at most $(n-1)\Phi$ time.

It takes p until $\sigma + (n-1)\Phi + \vartheta + \Phi$ until it sends its first $\langle \text{FIN}, s, x_p\rangle$, and $(n-1)\Phi$ additional time until it has completed its broadcast. (Alternatively p could also have entered round $r > s$ by some other means.) Any other G_i-correct process q, will have sent its FIN messages for round s by that time, or by either $\tau_g^i + \vartheta + (2n-1)\Phi$ or $\tau_g^i + \eta + n\Phi$ (by the argument given in the proof of Lemma 6.15). Let τ denote the maximum of these three times. As $\sigma \geq \tau_g^i$ we can ignore the second case and get

$$\tau \geq max \begin{cases} \sigma + \vartheta + (2n-1)\Phi \\ \tau_g^i + max\{\vartheta, \eta\} + n\Phi. \end{cases}$$

Then, by time $\tau + \Delta$, FIN messages from round s, or messages from later rounds have arrived at every G_i-correct process (including p) from all other G_i-correct processes. Since some process may be currently broadcasting the latest time when all G_i-correct processes have taken a reception step in which they gather sufficient evidence to enter round $r+1$ is guaranteed to occur before $\tau + \Delta + n\Phi$ only. Thus, we obtain the bound for σ' given in the lemma. □

Lemma 6.18 (Fastest Progress)**.** *Let σ be the minimal time where a σ-correct processes is in round s. Then no $[\sigma, \sigma + \vartheta)$-correct process can enter round $s+1$ before $\sigma + \vartheta$.*

Proof. Correct processes can only enter round $s+1$ by increasing r_{next} in line 26; entering round $s+1$ by any other means would violate the assumption that σ is the minimal time where a processes is in round s. Thus, any correct process needs $n-t$ $\langle \text{FIN}, s, x\rangle$ messages. But before sending these messages, correct processes have to wait for the timeout ϑ to expire after entering round s at time σ. □

While the last few lemmas were primarily concerned with how the round numbers of processes can be related to each other, we now turn to the set of values received in each of these rounds. Therefore our next lemmas are primarily related to the Rcv variable, which will then (Lemma 6.25 and following) be used to argue about the estimates x_p.

Lemma 6.19. *Let $\sigma \in G_i$ be the minimal time when there are $t+1$ G_i-correct processes in round s. If*

(i) $\vartheta > (3n-1)\Phi + 2\Delta$,
(ii) $\sigma \geq \tau_g^i + max\{\vartheta, \eta\}$,
(iii) $\sigma < \tau_b^i - \vartheta$, *and*
(iv) all G_i-correct processes are at least in round $s-1$ at σ.

then every G_i-correct process has round s entries in its Rcv entries for all G_i-correct processes before going to round $s+1$.

Proof. Let $\mathcal{S}$ denote the set of $t+1$ processes in s at σ, $\mathcal{S}' \subseteq \mathcal{S}$ those that were not in s before σ, and $\mathcal{R}$ those G_i-correct processes only in $s-1$ at time σ. Note that $|\mathcal{S} \setminus \mathcal{S}'| \leq t$, $|\mathcal{S}'| \geq 1$ (since σ is minimal), and $|\mathcal{R}| \geq n - |\mathcal{S}| - f(G_i)$ (by assumption (iv)). Since $\sigma \geq \tau_g^i + max\{\vartheta, \eta\}$, every process in $\mathcal{S} \setminus \mathcal{S}'$ has sent a round s message within G_i by $\sigma + n\Phi$ (due to an argument similar to that of Lemma 6.16). Processes in $\mathcal{S}'$ have finished sending their ROUND message for s by time $\sigma + (n-1)\Phi$.

Therefore, all G_i-correct processes have at least $t+1$ round s messages waiting to be received at $\sigma + n\Phi + \Delta$. Thus also going to round s (at the latest) in the first reception step following that time, which may be up to $n\Phi$ time later, as a process may have started an broadcast just before $\sigma + n\Phi + \Delta$.

Thus by $\tau = \sigma + 2n\Phi + \Delta$, all processes in $\mathcal{R} \cup \mathcal{S} = \Pi \setminus \mathcal{F}(G_i)$ have received a message from a process in $\mathcal{S}$, while being in round s or $s-1$ and thus add the value from these messages to $Rcv[s]$. By $\tau + (n-1)\Phi$ all processes in $\mathcal{R}$ are guaranteed to have sent their ROUND messages, which will be received in the first reception step after $\tau' = \tau + (n-1)\Phi + \Delta$. Thus, we are done, if we can show that no process in $\mathcal{S}$ has entered some round $r > s$ before this time, in particular, we show that no G_i-correct process can be in round $s+1$ by that time. Since $\mathcal{S}' \cup \mathcal{R}$ have entered round s at or after σ, and since $\vartheta > (3n-1)\Phi + 2\Delta \geq \tau' - \sigma$, the processes in $\mathcal{S}' \cup \mathcal{R}$ are still waiting for their ϑ timeout to expire, and thus no process can have more than $2t$ messages which are either a FIN message for round s, or some message for some later round. Therefore, no process can have received $n - t > 2t$ messages which would allow a G_i-correct process to pass the condition in line 26.

Thus we have shown that all G_i-correct processes hear from all other G_i-correct processes while being in round s and during $[\sigma, \sigma + \vartheta)$, with ϑ as in (i). Moreover, due to (iii) this interval lies within G_i. □

We now prove that every good period must have (at least) one combination of s and σ so that Lemma 6.19 can be applied.

Lemma 6.20. *Assume $W > 3\vartheta + (7n-4)\Phi + 3\Delta$ with $\vartheta > (3n-1)\Phi + 2\Delta$ and $\vartheta > \eta$. For each good period G_i there is at least one $\sigma \in G_i$ and a suitable s such that Lemma 6.19 can be applied.*

Proof. Lemma 6.19 considers a time σ, where for the first time $t+1$ processes are in some round s, given that:
(1) $\vartheta > (4n-1)\Phi + 2\Delta$,
(2) $\sigma \geq \tau_g^i + max\{\vartheta, \eta\}$,
(3) $\sigma < \tau_b^i - \vartheta$, and
(4) all G_i-correct processes are at least in round $s-1$ at σ.

We fix some G_i, and explore what happens after τ_g^i, in order to arrive at an time which fulfills all the above conditions. Note that, by the assumptions of the lemma, (2) is satisfied if $\sigma \geq \tau_g^i + \vartheta$ and (1) holds.

In the simpler case where all processes are in the same round r at τ_g^i, Lemma 6.15 entails that all processes are in round $r+1$ at the latest by $\tau_g^i + max\{\vartheta, \eta\} + (3n-2)\Phi + \Delta$. Surely, at some point in time σ between τ_g^i and this time such that before σ less than $t+1$ G_i-correct processes are in round $r+1$, but at least $t+1$ are in round $r+1$ at σ. If this time is after $\tau_g^i + \vartheta$, as required by (2), then (4) holds as well as all G_i-correct processes are already in r since τ_g^i.

If σ lies before $\tau_g^i + \vartheta$, then we know that at most $(2n-1)\Phi + \Delta$ later, that is by $\tau < \tau_g^i + \vartheta + (2n-1)\Phi + \Delta$ all G_i-correct processes have gone to round $r+1$ (since they have received the $\langle \text{ROUND}, r+1, \_\_ \rangle$ messages from the aforementioned at least $t+1$ processes. Now, we can apply Lemma 6.17 to τ to conclude that there is some $\sigma' \leq \tau + \vartheta + (3n-1)\Phi + \Delta$, where at least $t+1$ G_i-correct processes are in round $r+2$ for the first time. By, Lemma 6.18 this time cannot be before $\tau_g^i + \vartheta$ as well, therefore (2) holds. For (4), we have to show that at σ' all processes are at least in round $r+1$, that is, that it is guaranteed that $\sigma' \geq \tau$, which follows since we know from the above discussions that

$$\tau \leq \sigma + (2n-1)\Phi + \Delta \qquad \text{and} \qquad \sigma' \geq \sigma + \vartheta.$$

Thus we obtain, (using $\vartheta > (3n-1)\Phi + 2\Delta$),

$$\sigma' - \tau \geq \vartheta - ((2n-1)\Phi + \Delta) > n\Phi + \Delta \geq 0.$$

So we are done if $\sigma' < \tau_b^i - \vartheta$, i.e., if (3) holds as well, which will follow from the more general case below.

In case not all G_i-correct processes are in the same round at τ_g^i, we denote by $r+1$ the maximal round a G_i-correct process is in. It then follows from Lemma 6.16, that all processes are in round r by $\tau_g^i + \vartheta + (2n-1)\Phi + \Delta$. And by Lemma 6.17 that all G_i-correct processes must be in round $r+1$ at $\tau_g^i + 2\vartheta + (5n-2)\Phi + 2\Delta$. Clearly, the first time (in the following, denoted by σ) when (at least) $t+1$ processes were in round $r+1$ must be at or before this time.

When $\sigma \geq \tau_g^i + \vartheta + (2n-1)\Phi + \Delta$, then (2) and (4) clearly hold. Otherwise, we can determine points in time $\tau \leq \sigma(2n-1)\Phi + \Delta$ and $\sigma' \leq \tau + \vartheta + (3n-1)\Phi + \Delta$ by the same argument as above. Moreover, (2) and (4) hold by the argument above. Such that it only remains to show that (3) holds in this case, i.e., that $\sigma' \leq \tau_b^i - \vartheta$. Note that this will also imply (3) for the simpler case above, where the construction of σ' started from a smaller σ. Starting from

$$\sigma \leq \tau_g^i + \vartheta + (2n-1)\Phi + \Delta$$

we obtain

$$\tau \leq \tau_g^i + \vartheta + (2n-1)\Phi + \Delta + (2n-1)\Phi + \Delta = \tau_g^i + \vartheta + (4n-2)\Phi + 2\Delta$$

and in turn

$$\begin{aligned}\sigma' &\leq \tau_g^i + \vartheta + (4n-2)\Phi + 2\Delta + \vartheta + (3n-1)\Phi + \Delta \\ &= \tau_g^i + 2\vartheta + (7n-4)\Phi + 3\Delta, \\ &< \tau_b^i - W + 2\vartheta + (7n-4)\Phi + 3\Delta.\end{aligned}$$

Thus, (3) holds if

$$W > 3\vartheta + (7n-4)\Phi + 3\Delta,$$

which it does by definition. Thus (3) holds for the second case, and as the σ' in this case was after the one from the first case in the first case as well. □

For the rest of this section we implicitly assume $W > 3\vartheta + (7n-4)\Phi + 3\Delta$, and $\vartheta > \eta$. From Lemmas 6.19 and 6.20 we get the following two corollaries:

Corollary 6.21. *In each good period there is one round such that every G_i-correct process has a* Rcv *entry for that round containing a value for every G_i-correct process.*

Corollary 6.22. *Let $\mathcal{I} = [\tau_s, \tau_e) \subseteq G_i$, with $|\mathcal{I}| > 3\vartheta + (7n-4)\Phi + 3\Delta$ and $f(\mathcal{I}) = 0$. Then every process hears from every other process in at least one round r during $\mathcal{I}$.*

Moreover, it is interesting to observe what happens after we have applied Lemma 6.19 once. We know that at least $t+1$ processes are in the FIN-phase of round s at $\sigma + \vartheta$, and we can see when assuming a sufficiently long good period from the proof that all G_i-correct processes are in round s at $\sigma + (n-1)\Phi + \Delta + n\Phi$ at the latest, and start sending their ROUND messages for round s. From Lemma 6.17 we know that it takes at most $\vartheta + (3n-1)\Phi + \Delta$ to go from round s to round $s+1$. Since the prerequisites of Lemma 6.19 are now always true (recall the construction of σ' in Lemma 6.20), we can infer that the following Lemma 6.23 holds.

Lemma 6.23. *Let $\mathcal{I} \subseteq G_i$ be a good period of length $W + k(\vartheta + (3n-1)\Phi + \Delta)$, then every G_i-correct process goes through $k+1$ consecutive rounds during $\mathcal{I}$ in which it hears from every G_i-correct process.*

This in turn implies that rounds of different bad periods are disjoint (except for recovering processes, which may be in round 0 in both cases):

Corollary 6.24. *Consider two bad periods B_i and B_j with $i \neq j$, and let R_i (and R_j) be the set of rounds that $E(B_i)$-correct ($E(B_j)$)-correct, resp.) processes are in during the bad periods Then $(R_i \cap R_j) \subseteq \{0\}$.*

As we have now shown the properties of the round system our algorithm establishes, we turn our attention to the consensus core. The next two lemmas show that once a value is preferred by a large enough majority of (currently correct) processes, then this majority will persist across good and bad periods.

Lemma 6.25. *Consider two consecutive bad periods B_i and B_{i+1} and assume $n-t$ $E(B_i)$-correct processes p have the same value $x = v$ at the start of the common good period $G_{i+1} = E(B_i) \cap E(B_{i+1})$. Then at the start of B_{i+1}, all $E(B_{i+1})$-correct processes q have $x_q = v$.*

Proof. Because of Lemma 6.20, there is one round r_0, so that every G_i-correct process hears from every G_i-correct process in that round at some time in G_i. Then every G_i-correct process updates x in line 30, and since $n-t$ is a majority, to v. Since every $E(B_{i+1})$-correct process is also G_i-correct, the lemma holds. □

Lemma 6.26. *Assume $n-t$ B_i-correct processes have the same value $x_p = v$ at some $\tau \in B_i$, then they still have this value at the start of the following good period G_{i+1}.*

Proof. Assume some process p updates x_p in B_i. Then by line 29 p has received $n-t$ messages for some round number r. Because of Corollary 6.24, all messages in Rcv for this round stem from the current bad period. Thus at least $n-2t$ of them have been sent by correct processes. Because of $n > 5t$, this is true for no other value than $\bar{x}$. □

We now have all the pieces necessary for proving the properties of the consensus core of our algorithm.

Lemma 6.27 (Agreement)**.** *If* $n > 5t$*, two processes* p *and* q *decide* v_p *and* v_q *at times* τ_p *and* τ_q *respectively, and* $p \in \mathcal{C}(\tau_p) \wedge q \in \mathcal{C}(\tau_q)$*, then* $v_p = v_q$*.*

Proof. W.l.o.g., assume that $\tau_p \leq \tau_q$. Since p decided v_p, it must have received at least n messages containing that value for one round. Thus all $\mathcal{I}$-correct processes must have sent this value as their estimate x, where $\mathcal{I}$ is the period in which p decides. Thus at τ_p all $\mathcal{I}$-correct processes p' must have $x_{p'} = v_p$. Now Lemmas 6.25 and 6.26 imply that at the time(s) when the messages are sent that cause q to decide, all processes correct in that (enclosing) period have v_p as their estimate x. Consequently, v_q can only be v_p. □

Lemma 6.28 (Validity)**.** *If* $n > 5t$ *and all initial values are* v *and if* $p \in \mathcal{C}(\tau_p)$ *decides* v_p *at time* τ_p*, then* $v_p = v$*.*

Proof. Either time 0 is the beginning of a good period, or in a bad period. In either case, Lemma 6.25 and Lemma 6.26, imply that no process will ever take another value $v' \neq v$ in any round $r > 0$, thus no other value can be decided. □

Lemma 6.29 (Termination)**.** *If there is a good period* G_i *containing a subinterval* $\mathcal{I}$ *of length* $W + \vartheta + (3n - 1)\Phi + \Delta$ *in which all processes are correct, then every process decides.*

Proof. From Lemma 6.23 we know that there are two (consecutive) rounds with every process hearing from every process, therefore when processes go through lines 29–32 for the first time all processes set their x to the same value (in line 30) since all receive the same set of messages.

This results in every process proposing the same value in the second round, and therefore deciding in line 32. □

When all processes are correct, have the same value, and consensus is started synchronously at all processes and in a good period (i.e., an *initial good period*), termination is much faster: we can apply the arguments of Lemma 6.19 to the starting time[9] which reveals that all processes hear their $\langle \text{ROUND}, 1, x \rangle$ messages within the ϑ timeout. The duration from this round follows from Lemma 6.17. Therefore we get,

Corollary 6.30 (Fast decision)**.** *If there is an initial good period of duration* $\vartheta + (3n - 1)\Phi + \Delta$ *in which all processes boot simultaneously there are no faulty processes and all processes initially agree on some value* v*, then processes decide within one round.*

6.3.4. Extensions

In this section we briefly discuss some extensions that can be applied to our algorithm to increase its efficiency.

Multiple instances. First we illustrate how our algorithm can be modified to work for several instances of consensus. The straightforward approach would be of course to use an extra instance of Algorithm 6.2 for each consensus. This is, however, not necessary. In fact, the same round structure can be shared by all instances. This stems from the fact that an execution does not have to start in round 1, since the algorithm allows any number of rounds where "nothing" happens, any new instance of consensus just starts in the current round of the algorithm. The only variables that need thus to be duplicated for each

[9] The lower bound on σ is not necessary in this case as $S = S' = \Pi$.

instance are the decision value and the array $\mathtt{Rcv}$. For the latter, another optimization is possible. Obviously only the value of the current round and the following round are needed by the algorithm. Thus the size of this variable reduces to $2n$ entries per instance.

Additional permanent crashes. Another fairly straightforward improvement is to allow permanent crash faulty processes as well, which also allows decisions at times when some processes are still down. This can be achieved by setting E (recall Algorithm 6.1) to a value smaller than n. It is easy to see that if we want to tolerate t_c crashes in addition to t Byzantine/recovery faults, we need to set the threshold $T = n - t - t_c$ and the threshold $E = n - t_c$. The resulting resilience bound $n > 5t + 3t_c$ can then be deduced from the formulas in Section 6.2.

This extension is also interesting with respect to the work of Widder et al. [2007], who consider a model, where the system is synchronous and Byzantine processes eventually crash but do never recover. In this case $n > 2f+1$ suffices for solving consensus[10] [Widder et al. 2007]. When one considers a model which allows every Byzantine processes to either eventually remain dead forever, or to eventually restart and behave well, then our algorithm allows solving consensus when $n > 8t$.

Fast decision. Our algorithm is already fast in the classical sense, i.e., if all processes are correct, have the same value and consensus is started at the same time at all processes, it terminates within one round. However, this round might take $\vartheta + (3n - 1)\Phi + \Delta$ time. A simple modification makes the algorithm really fast without invalidating any of our results: in the first round, if a process receives all n identical ROUND messages, it can immediately decide on that value. This can be achieved by executing the transition function part (lines 29 to 32) for the first round as soon as one has received a message from every other process. This modification allows a decision immediately after the *actual* transmission delay, which might be much smaller than Δ. This behaviour is called weakly one-step in [Song and Renesse 2008], where it is also shown that $n = 5t + 1$ is the lower bound for such algorithms, thus our solution is optimal in this respect.

For strongly fast algorithms, i.e., those that are fast even when up to t processes behave Byzantine from the start but all correct processes share the same initial value, a lower bound of $n = 7t + 1$ is known.[Song and Renesse 2008] We conjecture that our algorithm could also be made strongly one-step quite easily. In order to do so we would have to change the algorithm such that it decides when it receives $n - 2t$ messages with the same value ([Song and Renesse 2008] shows that this is a necessary condition for every strongly fast algorithm). When setting $E = n - 2t$ the conditions on n, T, and E lead to the requirement that $n > 9t$, which reveals that the resulting algorithm to be suboptimal in this respect.

[10] Note, that that algorithm is not fast.

PART IV

THE END

CHAPTER 7

SUMMARY AND DISCUSSION OF RESULTS

MOST EARLY WORK ON consensus algorithms relies on the paradigm of Byzantine processes, i.e., on *permanent* and *static* value faults. In the seminal papers [Pease et al. 1980; Lamport et al. 1982], which spawned research on the consensus problem, even synchrony was considered static, that is a *perpetually* synchronous system was assumed. It is (more or less) only this last aspect that has been relaxed, to also allow non-perpetual synchrony assumptions, leading to partially synchronous systems. While [Dwork et al. 1988] considered intermittent periods of synchrony[1], the whole failure detector community which emerged from the works of Chandra and Toueg [1996] assumes that eventually there is perpetual synchrony. Indeed, it is impossible to solve any interesting distributed computing problem in the asynchronous system augmented by intermittently reliable failure detectors, as the processes may simply not take any steps in the "good" periods [Charron-Bost et al. 2008, see 3.2 Time contraction].

In contrast to the work mentioned above the aim of this thesis is to investigate models of a more *dynamic* nature. Thus we are interested in *transient* and *mobile* failures, as well as systems with *intermittent* synchronous periods.

In Chapter 3 we have analyzed the relations of different synchrony assumptions. We have done so by showing that it is possible to transform algorithms for eventually synchronous models. However, as we have already discussed before our notion of efficiency-preserving transformations allows to reason about variants of these models which are synchronous only intermittently. In summary, unbounded transformations or unbounded shift allows for a similar effect as the time contraction [Charron-Bost et al. 2008] for failure detectors: It is possible that, from the "viewpoint" of the algorithm the system has no synchronous periods at all.

We have then turned our attention to failures: here we first concentrated on dynamic communication failures. Regarding these, it is clear that allowing an unlimited number of faults renders every "interesting" distributed computing problem trivially impossible —

[1] They assume eventual synchrony but then determine the number of steps needed to terminate the algorithm, thereby establishing how long the intermittent synchronous period must be in order to allow the algorithms to decide.

even if they are not dynamic. Moreover, Santoro and Widmayer [1989] have shown that synchrony does not really help: when up to $n-1$ (or $\lfloor \frac{n}{2} \rfloor$ arbitrary) link failures can occur in a round consensus becomes impossible to solve. The problematic case occurs when all the faults affect the outgoing links of one process. (This may happen to a different process in every round.) One approach to get around this impossibility, is to appropriately limit the number of faults that can occur system wide in each round [Pinter and Shinahr 1985; Santoro and Widmayer 1990]. This approach has the drawback that only $O(n)$ faulty links can be tolerated. In the course of this thesis we have seen two alternatives.

On the one hand (Chapter 5), we have seen that one can mask up to $n \cdot \ell^a$ total arbitrary link failures per round, by restricting the number of links that can occur per round and per process for incoming and outgoing links to $\ell^a < \lfloor \frac{n}{4} \rfloor$. When in addition to this we allow up to ℓ^o links (per round and per process) to be omissive, we need $n > 4\ell^a + 2\ell^o$ to mask all link failures. The impossibility of Chapter 4 implies that these results are optimal.

On the other hand, $\mathcal{A}_{T,E}$ (Chapter 6) also allows up to $n^2/4$ arbitrary transmission faults per round but there is no perpetual limit on the number of purely omissive links. We have shown that consensus can be solved under these conditions. Note that the apparent contradiction with the lower bound of Theorem 4.7 is void. This is due to the fact that in the analysis of $\mathcal{A}_{T,E}$ we distinguish strongly between the safety and liveness of the consensus algorithm: For $\mathcal{A}_{T,E}$'s safety it is sufficient that less than $n/4$ corruptions per process and round occur. But in order to ensure termination, two rounds are necessary where the assumptions are much higher than in the lower bound (cf. predicate $\mathcal{P}^{\mathcal{A},live}$ in Figure 6.1 on page 75).

The aforementioned separation between safety and liveness of communication has another interesting aspect: Guarantees about the liveness of communication, that is (in this case) guarantees about the arrival of messages within a round do not only require links to be reliable, but also timely. In the implementation of the round model "around" $\mathcal{A}_{T,E}$ in Chapter 6.3, we have introduced an example where the system was timely only sometimes. This was shown to be sufficient to solve consensus since $\mathcal{A}_{T,E}$ only requires for communication to be live in some rounds. (Other algorithms, e.g., $\mathcal{U}_{T,E,\alpha}$ [Biely et al. 2007a], require some liveness of communication in all rounds.)

Another example of a round-based model with dynamic failures is the perception based fault model on which we based our modelling in Chapters 4 and 5. In fact that model is slightly more detailed than the (restricted) model we used. Although it was originally proposed for synchronous (round-based) systems [Schmid and Weiss 2002], it has been generalized to a synchrony model where processes do not run in lockstep and only know the ratio of the upper and the lower bound on the end-to-end communication delay [Widder and Schmid 2007]. (Note that failures are still defined based on rounds here.) The perception based fault model belongs to the class of hybrid fault models and allows for a wide range of types of failures for processes: (clean) crash faults, (asymmetric) omissions, symmetric (value) faults and arbitrary (i.e., Byzantine) faults. As for links it too considers dynamic omissive and arbitrary faulty links. This model has been successfully used in conjunction with different models of synchrony to analyze algorithms for a wide range of distributed computing problems, agreement [Biely 2002; Biely and Schmid 2001; Schmid et al. 2002; Weiss and Schmid 2001; Weiss 2002] and Synchronization [Widder 2004; Widder and Schmid 2007]. All the failure classes are defined by how they affect different message matrices, that is there are (for each round) matrices of the

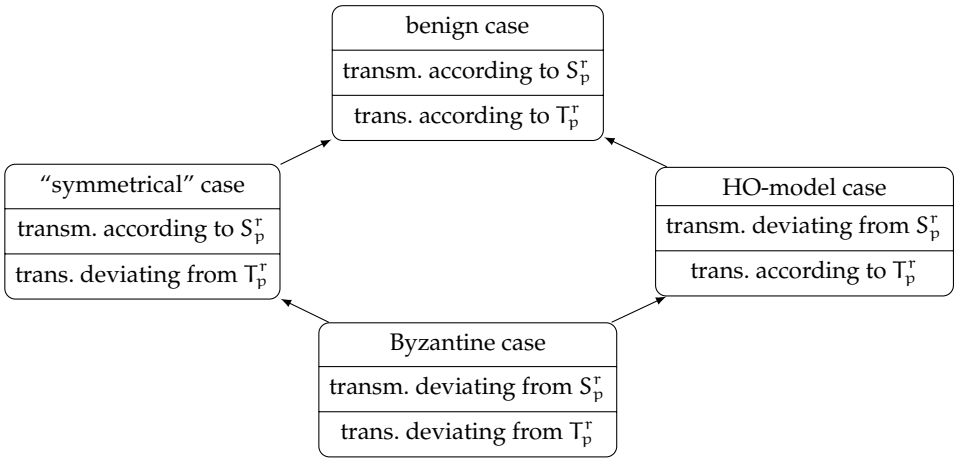


Figure 7.1. *Possible types of corruption*

messages created, actually sent and received.

Returning to the HO-model, we note that the approach taken to model failures is somewhat different from the perception based model: The definition of the *HO* sets refers to the values received only (recall $\vec{\mu}_p^r$), while the *SHO* sets are defined with respect to the difference between these $\vec{\mu}_p^r$ vectors and the values that should have been sent. So far one could consider the approach similar, as the reception matrix of the perception based fault model is — in fact — equivalent to the matrix formed by the $\vec{\mu}_p^r$ vectors of all processes. This strong similarity allows us to actually express the link failure assumptions of the perception based failure model in terms of the heard-of sets $HO(p, r)$ and talk-to[2] sets $TT(p, r)$ of all processes (and their safe variants). Note that the assumptions are the same for all processes and for all rounds, while the *HO* approach allows for much richer predicates (recall, e.g., Figure 6.1).

Moreover, with the *HO*-approach the reason why a certain transmission is not present or different from the message which ought to be sent is in fact hidden from the algorithm designer, by only having guarantees about predicates over the *HO* and *SHO* sets. For this view it is essential that the internal state of the HO-machine is never corrupted, which leads us to our next point: Examining how an HO machine, in general, can suffer from corruption. The model at the top of Figure 7.1, where no value faults occur, is the benign case. In the model at the bottom, transitions might deviate from what is prescribed by transition functions and transmissions might deviate from what is prescribed by sending functions in an execution. This corresponds to the classical Byzantine process failure model assumption [Pease et al. 1980; Lamport et al. 1982] and also that of Section 6.3.

Naturally, there are two models that lie in between these two extreme cases. In the model on the left of Figure 7.1 transmissions always follow the sending functions, but state transitions might not follow the transition functions (thus also state corruptions may occur). Consequently, by designing (broadcast based) algorithms that send the same message to all other processes, faulty behavior can be restricted to "symmetric failures" [Schmid et al. 2002] (also termed "identical Byzantine" in the book by Attiya and Welch [2004]). In practice such behavior can be implemented (if not present a priori) with *signed messages*. Thus, signatures are an important implementation concept in order to ensure

[2]The talk-to sets are defined as $TT(p, r) := \{q \in \Pi \mid p \in HO(q, r)\}$.

such behavior. However, given the lack of a formal definition of signatures (cf. Lynch's comment [Lynch 1996, p. 115]), we believe that signatures have to be seen as an engineering tool in the context of predicate implementations but should be kept separate from the formal aspects of distributed computing.

Finally, the model on right side of Figure 7.1 is the one considered in Section 6.2 (and also the one of Santoro and Widmayer [1989; 1990], as well as [Biely et al. 2007a]). One could argue that this approach is only of theoretical, but not of practical interest, given that techniques like *signatures* or *error correcting codes* could be used to transform value into benign failures (on an engineering level). However, as the predicates do not limit the sending function itself considerably it is indistinguishable to other processes (or in general any an outside observer) whether such a process has a corrupted state or not for.[3] For instance the synchronous system with reliable links and at most f Byzantine processes can be characterized by a predicate on the global safe kernel $SK := \bigcap_{r\in\mathbb{N}} \bigcap_{p\in\Pi} SHO(p,r)$ [Biely et al. 2007a]:

$$|SK| \geq n - f.$$

As another example, the predicate corresponding to the asynchronous system with reliable links and at most f Byzantine processes, defined in terms of the altered span $AS := \bigcup_{r\in\mathbb{N}} AS(r)$:

$$\forall p \in \Pi : \ \forall r > 0 : \ |HO(p,r)| \geq n - f \ \wedge \ |AS| \leq f. \tag{7.1}$$

Thus, any result established for, for instance an *HO*-machine with the predicate (7.1) is also valid for the the corresponding system models, that is, the asynchronous Byzantine model mentioned above.

In sharp contrast to the approach taken with $\mathcal{A}_{T,E}$ and the HO-approach in general — that is, modelling process failures solely by the faulty transmissions they cause — other models [Dolev and Strong 1982a; Gong et al. 1995; Perry and Toueg 1986] map link faults onto either the sending or receiving process. Apart from the obvious difficulty to model dynamic link failures in such settings, the major deficiency of such models is that the number of link faults tolerable by this approach is usually extremely low compared to models that deal with link faults explicitly. Consider, for instance, a system of four processes $n = 4$ where each process p_x is hit by only one benign link fault that hits the link to $p_{((x+1) \bmod 4)}$. It is obvious that every process has to be considered faulty in the models mentioned above. Even when using the general omission model (where faulty processes may fail to send and receive) does not really help although one could attribute the link failures to only two processes, in the former case. In addition, now one faces the problem that (in the example above) there are two sets of processes that can be considered faulty. Clearly, exempting the faulty processes from fulfilling the properties of consensus does not make sense under such circumstances. Considering uniform consensus[4] instead of consensus does not help, as it is impossible to achieve in such systems unless $n > 2f$ [Neiger and Toueg 1990] which is not the case in our example. In contrast, with the perception based approach (cf. Chapters 4 and 5) one would just set $\ell = 1$ and be able to solve consensus (recall, that ℓ denotes the number of links that may be faulty per round for each process.)

[3] A similar point was also made by [Lampson 2001].

[4] Recall that, in this variant, Agreement has to hold for the faulty processes as well; faulty processes are not required to terminate, however.

Moreover, since links are not required to move in our model — in fact the proofs do not use the fact that link failures may move from one link to another from round to round — the result also holds for systems where link failures are static [Siu et al. 1998b; Dolev et al. 1995; Goldreich and Sneh 1992; Sayeed et al. 1995]. If we restrict our considerations to (permanent) omissions, we can interpret our results in this regard as minimum connectivity results [Meyer and Pradhan 1991; Dolev 1982; Hadzilacos 1987]. A brief survey of a wider range of problems solvable and unsolvable in systems where links may be faulty in a number of ways can be found in [Fich and Ruppert 2003, Section 6.4].

Regarding the combination of Byzantine failures and partially synchronous systems the seminal work of Dwork et al. [1988] has to be noted. Besides the work of [Aguilera et al. 2006] which considers quite weak synchrony assumptions (but higher link reliability assumptions than our $\mathcal{A}_{T,E}$ implementation) research seems to be concentrating on Byzantine variants of the famous Paxos algorithm (e.g., [Castro and Liskov 2002; Lampson 2001; Abraham et al. 2006; Zieliński 2004]). There have been several approaches to make these Paxos related approaches more practical[5] [Rodrigues et al. 2001; Cowling et al. 2006; Kotla et al. 2007]. Martin and Alvisi [2006] have integrated Byzantine fault-tolerance of Byzantine Paxos with the "fastness" of fast Paxos[Lamport 2005]. Here we should note that while some of the mentioned research considers itself to be on asynchronous consensus, it is in fact not. Rather all these approaches consider eventually synchronous systems.

The problem definition of the Byzantine Paxos literature is closer related to Byzantine agreement[6] than to consensus, and thus it is closely related to the notion of faulty processes as integrity only restricts delivery of messages in the case of a *correct* transmitter. This poses a problem when comparing these approaches to the HO model, where there is not notion of faulty or correct transmitters. Still, fast Byzantine Paxos shares some algorithmic similarities with $\mathcal{A}_{T,E}$ and (by transitivity) also with our lower level implementation with dynamic Byzantine failures.

None of the aforementioned papers consider a consensus algorithm with recovery in the Byzantine case, except for the replication protocol of BFT [Castro and Liskov 2002] which uses a variant of proactive recovery. However, in their case, recovery may occur only between several consensus instances. For a single instance of consensus, BFT has a fixed set of faulty processes. Moreover, their approach heavily relies on cryptography (recall the comments on authentication above), so that a process after recovery enters the system with a new identity, makes use of stable storage and tries to verify whether this stored state is still valid. It also uses extra recovery messages to integrate a new replica into the system. This is in contrast to the $\mathcal{A}_{T,E}$ implementation, where we do not need any of these means. Our variant of $\mathcal{A}_{T,E}$ (Section 6.3) allows recoveries *during* a consensus instance and does not specify the reason for recoveries.

[5] It seems that Lamport is correct when he says: "Computer scientists in this field must keep up the pretense that everything they do is practical." [Lamport 2009]

[6] Recall, that in the Byzantine agreement problem, there is only one transmitter, and the problem is to reach consensus on the value sent.

PART

APPENDIX

REFERENCES

ABRAHAM, I., CHOCKLER, G., KEIDAR, I., AND MALKHI, D. 2006. Byzantine disk Paxos: optimal resilience with Byzantine shared memory. *Distributed Computing 18,* 5, 387–408.

AGUILERA, M. K., CHEN, W., AND TOUEG, S. 2000. Failure detection and consensus in the crash-recovery model. *Distributed Computing 13,* 2 (Apr.), 99–125.

AGUILERA, M. K., DELPORTE-GALLET, C., FAUCONNIER, H., AND TOUEG, S. 2003. On implementing Omega with weak reliability and synchrony assumptions. In *Proceeding of the 22nd Annual ACM Symposium on Principles of Distributed Computing (PODC'03)*. ACM Press, New York, NY, USA, 306–314.

AGUILERA, M. K., DELPORTE-GALLET, C., FAUCONNIER, H., AND TOUEG, S. 2004. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the 23th ACM Symposium on Principles of Distributed Computing (PODC'04)*. ACM Press, St. John's, Newfoundland, Canada, 328–337.

AGUILERA, M. K., DELPORTE-GALLET, C., FAUCONNIER, H., AND TOUEG, S. 2006. Consensus with Byzantine failures and little system synchrony. In *DSN '06: Proceedings of the International Conference on Dependable Systems and Networks*. IEEE Computer Society, Washington, DC, USA, 147–155.

ANCEAUME, E., DELPORTE-GALLET, C., FAUCONNIER, H., HURFIN, M., AND LE LANN, G. 2004a. Designing modular services in the scattered Byzantine failure model. In *3rd International Symposium on Parallel and Distributed Computing (ISPDC 2004)*. IEEE Computer Society, 262–269.

ANCEAUME, E., DELPORTE-GALLET, C., FAUCONNIER, H., HURFIN, M., AND LE LANN, G. 2004b. Modular reliable services with Byzantine recovery. Publication Interne 1602, IRISA. Feb.

ANCEAUME, E., DELPORTE-GALLET, C., FAUCONNIER, H., HURFIN, M., AND WIDDER, J. 2007. Clock synchronization in the Byzantine-recovery failure model. In *International Conference On Principles Of DIstributed Systems OPODIS 2007*. LNCS. Springer Verlag, Guadeloupe, French West Indies, 90–104.

ANGLUIN, D. 1980. Local and global properties in networks of processors (extended abstract). In *Proceedings of the Twelfth Annual ACM Symposium on Theory of Computing*. ACM, 82–93.

ANGLUIN, D., FISCHER, M. J., AND JIANG, H. 2006. Stabilizing consensus in mobile networks. In *Distributed Computing in Sensor Systems, Second IEEE International Conference, DCOSS 2006, San Francisco, CA, USA, June 18-20, 2006, Proceedings*, P. B. Gibbons, T. F. Abdelzaher, J. Aspnes, and R. Rao, Eds. Lecture Notes in Computer Science, vol. 4026. Springer, 37–50.

ARISTOTLE. *Physics*. Z9 239b 14ff. (cf. DK 29 A 26).

ATTIYA, H. AND WELCH, J. 2004. *Distributed Computing*, 2nd ed. John Wiley & Sons.

AZADMANESH, M. AND KIECKHAFER, R. M. 2000. Exploiting omissive faults in synchronous approximate agreement. *IEEE Transactions on Computers 49,* 10 (Oct.), 1031–1042.

AZADMANESH, M. H. AND KIECKHAFER, R. M. 1996. New hybrid fault models for asynchronous approximate agreement. *IEEE Transactions on Computers 45,* 4, 439–449.

BARAK, B., HALEVI, S., HERZBERG, A., AND NAOR, D. 2000. Clock synchronization with faults and recoveries (extended abstract). In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*. ACM Press, Portland, Oregon, United States, 133–142.

BERNSTEIN, P. A., HADZILACOS, V., AND GOODMAN, N. 1987. *Concurrency Control and Recovery in Database Systems*. Addison-Wesley Publishing Company.

BIELY, M. 2002. Byzantine agreement under the perception-based fault model, Diplomarbeit, Technische Universität Wien, Department of Automation.

BIELY, M. 2003. 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)*. Marina Del Rey, California, USA, 146–151.

BIELY, M. 2006. On the impact of link faults on Byzantine agreement. Research Report 48/2006, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 1-3/182-2, 1040 Vienna, Austria. (submitted).

BIELY, M., CHARRON-BOST, B., GAILLARD, A., HUTLE, M., SCHIPER, A., AND WIDDER, J. 2007a. Tolerating corrupted communication. In *Proceedings of the 26th ACM Symposium on Principles of Distributed Computing (PODC'07)*. ACM, Portland, OR, USA, 244–253.

BIELY, M. AND HUTLE, M. 2009. Consensus when all processes may be Byzantine for some time. In *International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009)*. (to appear).

BIELY, M., HUTLE, M., DRAQUE PENSO, L., AND WIDDER, J. 2007b. Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency. In *9th International Symposium on Stabilization, Safety, and Security of Distributed Systems*. Lecture Notes in Computer Science, vol. 4838. Springer Verlag, Paris, 4–20.

BIELY, M. AND SCHMID, U. 2001. Message-efficient consensus in presence of hybrid node and link faults. Tech. Rep. 183/1-116, Department of Automation, Technische Universität Wien. August.

BIELY, M., SCHMID, U., AND WEISS, B. 2009. Synchronous consensus under hybrid process and link failures. *(under submission)*.

BIELY, M. AND WIDDER, J. 2006. Optimal message-driven implementation of Omega with mute processes. In *Proceedings of the Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*. LNCS, vol. 4280. Springer Verlag, Dallas, TX, USA, 110–121.

BIELY, M. AND WIDDER, J. 2009. Optimal message-driven implementations of omega with mute processes. *ACM Transactions on Autonomous and Adaptive Systems 4,* 1, Article 4, 22 pages.

BIRAN, O., MORAN, S., AND ZAKS, S. 1990. A combinatorial characterization of the distributed 1-solvable tasks. *Journal of Algorithms 11,* 3, 420–440.

CASTRO, M. AND LISKOV, B. 2002. Practical byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems 20,* 4, 398–461.

CHANDRA, T. D., HADZILACOS, V., AND TOUEG, S. 1996. The weakest failure detector for solving consensus. *Journal of the ACM 43,* 4 (June), 685–722.

CHANDRA, T. D. AND TOUEG, S. 1996. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM 43,* 2 (March), 225–267.

CHARRON-BOST, B., GUERRAOUI, R., AND SCHIPER, A. 2000. Synchronous system and perfect failure detector: solvability and efficiency issues. In *Proceedings of the International Conference on Dependable System and Networks (DSN'00)*. IEEE Computer Society Press.

CHARRON-BOST, B., HUTLE, M., AND WIDDER, J. 2008. In search of lost time. Tech. Rep. LSR-REPORT-2008-006, EPFL.

CHARRON-BOST, B. AND SCHIPER, A. 2004. Uniform consensus is harder than consensus. *J. Algorithms 51,* 1, 15–37. Also published as Tech. Rep. DSC/2000/028, Ecole Polytechnique Fédérale de Lausanne.

CHARRON-BOST, B. AND SCHIPER, A. 2006a. The heard-of model: Unifying all benign failures. Tech. Rep. LSR-REPORT-2006-004, EPFL.

CHARRON-BOST, B. AND SCHIPER, A. 2006b. Improving fast Paxos: being optimistic with no overhead. In *12th IEEE Pacific Rim International Symposium on Dependable Computing (PRDC 2006)*. IEEE Computer Society, 287–295.

CHARRON-BOST, B. AND SCHIPER, A. 2007. Harmful dogmas in fault tolerant distributed computing. *SIGACT News 38*, 1, 53–61.

CHARRON-BOST, B. AND SCHIPER, A. 2009. The Heard-Of mode: computing in distributed systems with benign faults. *Distributed Computing*.

COWLING, J., MYERS, D., LISKOV, B., ES, R. R., AND SHRIRA, L. 2006. HQ replication: a hybrid quorum protocol for byzantine fault tolerance. In *Proceedings of the 7th symposium on Operating systems de sign and implementation (OSDI '06)*. USENIX Association, Berkeley, CA, USA, 177–190.

CRISTIAN, F. AND FETZER, C. 1994. Fault-tolerant internal clock synchronization. In *Proceedings of the Thirteenth Symposium on Reliable Distributed Systems*. Dana Point, Ca., 22–31.

CRISTIAN, F. AND FETZER, C. 1999. The timed asynchronous distributed system model. *IEEE Transactions on Parallel and Distributed Systems 10,* 6, 642–657.

DIJKSTRA, E. W. 1965. Solution of a problem in concurrent programming control. *Communications of the ACM 8,* 9, 569.

DIJKSTRA, E. W. 1974. Self-stabilizing systems in spite of distributed control. *Communications of the ACM 17,* 11, 643–644.

DOLEV, D. 1982. The Byzantine generals strike again. *Journal of Algorithms 3,* 1, 14–30.

DOLEV, D., DWORK, C., AND STOCKMEYER, L. 1987. On the minimal synchronism needed for distributed consensus. *Journal of the ACM 34,* 1 (Jan.), 77–97.

DOLEV, D., FISCHER, M. J., FOWLER, R., LYNCH, N., AND STRONG, H. R. 1982. An efficient algorithm for Byzantine agreement without authentication. *Information and Control 52*, 257–274.

DOLEV, D., HALPERN, J. Y., SIMONS, B., AND STRONG, R. 1995. Dynamic fault-tolerant clock synchronization. *Journal of the ACM 42,* 1, 143–185.

DOLEV, D., REISCHUK, R., AND STRONG, R. 1994. Observable clock synchronization. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing (PODC '94)*. ACM, New York, NY, USA, 284–293.

DOLEV, D. AND STRONG, H. R. 1982a. Distributed commit with bounded waiting. In *Proceedings of the 2nd Symposium on Reliability in Distributed Software and Database Systems,*. Pittsburgh, 53–60.

DOLEV, D. AND STRONG, H. R. 1982b. Polynomial algorithms for multiple processor agreement. In *Proceedings 14th Annual ACM Symposium on Theory of Computing (STOC'82)*. San Francisco, 401–407.

DOLEV, S., ISRAELI, A., AND MORAN, S. 1997. Resource bounds for self-stabilizing message-driven protocols. *SIAM Journal on Computing 26,* 1, 448–458.

DOLEV, S., KAT, R. I., AND SCHILLER, E. M. 2006. When consensus meets self-stabilization. In *Proceedings of the International Conference On the Principles of Distributed Systems (OPODIS'06)*. Lecture Notes in Computer Science, vol. 4305. Springer, 45–63.

DOLEV, S. AND WELCH, J. L. 2004. Self-stabilizing clock synchronization in the presence of Byzantine faults. *Journal of the ACM 51,* 5, 780–799.

DUTTA, P., GUERRAOUI, R., AND KEIDAR, I. 2007. The overhead of consensus failure recovery. *Distributed Computing 19,* 5-6, 373–386.

DUTTA, P., GUERRAOUI, R., AND LAMPORT, L. 2005. How fast can eventual synchrony lead to consensus? In *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN'05)*. Los Alamitos, CA, USA, 22–27.

DWORK, C., LYNCH, N., AND STOCKMEYER, L. 1988. Consensus in the presence of partial synchrony. *Journal of the ACM 35,* 2 (Apr.), 288–323.

DWORK, C. AND SKEEN, D. 1984. Patterns of communication in consensus protocols. In *Proceedings of the third annual ACM symposium on Principles of distributed computing (PODC '84)*. ACM, New York, NY, USA, 143–153.

EINSTEIN, A. 1905. Zur Elektrodynamik bewegter Körper. *Annalen der Physik 322,* 10, 891–921.

ELRAD, T. AND FRANCEZ, N. 1982. Decomposition of distributed programs into communication-closed layers. *Science of Computer Programming 2,* 3, 155–173.

FETZER, C., SCHMID, U., AND SÜSSKRAUT, M. 2005. On the possibility of consensus in asynchronous systems with finite average response times. In *Proceedings of the 25th International Conference on Distributed Computing Systems (ICDCS'05)*. IEEE Computer Society, Washington, DC, USA, 271–280.

FICH, F. AND RUPPERT, E. 2003. Hundreds of impossibility results for distributed computing. *Distributed Computing 16*, 121–163.

FISCHER, M. J. 1983. The consensus problem in unreliable distributed systems (a brief survey). In *Fundamentals of Computation Theory*. LNCS, vol. 158. Springer Verlag, 127–140.

FISCHER, M. J. AND LYNCH, N. 1982. A lower bound for the time to assure interactive consistancy. *Information Processing Letters 14,* 4 (May), 198–202.

FISCHER, M. J., LYNCH, N., AND MERRITT, M. 1986. Easy impossibility proofs for distributed consensus problems. *Distributed Computing 1,* 1, 26–39.

FISCHER, M. J., LYNCH, N. A., AND PATERSON, M. S. 1985. Impossibility of distributed consensus with one faulty process. *Journal of the ACM 32,* 2 (Apr.), 374–382.

GAFNI, E. 1998. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*. ACM Press, Puerto Vallarta, Mexico, 143–152.

GARAY, J. A. AND MOSES, Y. 1993. Fully polynomial Byzantine agreement in $t + 1$ rounds. In *Proceedings of the 25th Annual ACM Symposium on Theory of Computing*. San Diego, California, USA, 31–41.

GÄRTNER, F. C. AND PLEISCH, S. 2002. 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*. Springer-Verlag, 280–294.

GOLDREICH, O. AND SNEH, D. 1992. On the complexity of global computation in the presence of link failures: The case of uni-directional faults. In *Symposium on Principles of Distributed Computing*. 103–111.

GONG, L., LINCOLN, P., AND RUSHBY, J. 1995. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In *Proceedings Dependable Computing for Critical Applications-5*. Champaign, IL, 139–157.

GOPAL, A. S. 1992. Fault-tolerant broadcasts and multicasts: the problem of inconsistency and contamination. Ph.D. thesis, Cornell University, Ithaca, NY, USA.

GRAY, J. N. 1978. Notes on data base operating systems. In *Operating Systems: An Advanced Course*, G. S. R. Bayer, R.M. Graham, Ed. Lecture Notes in Computer Science, vol. 60. Springer, New York, Chapter 3.F, 465.

GUERRAOUI, R. 1995. Revisiting the relationship between non blocking atomic commitment and consensus problems. In *Distributed Algorithms (WDAG-9)*. Springer Verlag (LNCS).

HADZILACOS, V. 1987. Connectivity requirements for Byzantine agreement under restricted types of failures. *Distributed Computing 2*, 95–103.

HADZILACOS, V. AND TOUEG, S. 1993. Fault-tolerant broadcasts and related problems. In *Distributed Systems*, 2nd ed., S. Mullender, Ed. Addison-Wesley, Chapter 5, 97–145.

HOCH, E., DOLEV, D., AND DALIOT, A. 2006. Self-stabilizing Byzantine digital clock synchronization. In *Proceedings of the Eighth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2006)*. LNCS, vol. 4280. Springer Verlag, Dallas, TX, USA, 350–362.

HUTH, M. AND RYAN, M. 2004. *Logic in Computer Science: Modelling and Reasoning about Systems*. Cambridge University Press.

HUTLE, M., MALKHI, D., SCHMID, U., AND ZHOU, L. 2009. Chasing the weakest system model for implementing omega and consensus. *IEEE Transactions on Dependable and Secure Computing*. (to appear).

HUTLE, M. AND SCHIPER, A. 2007a. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In *37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 92–101.

HUTLE, M. AND SCHIPER, A. 2007b. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'07)*. IEEE Computer Society, Edinburgh, UK, 92–101.

HUTLE, M. AND WIDDER, J. 2005. On the possibility and the impossibility of message-driven self-stabilizing failure detection. In *Proceedings of the Seventh International Symposium on Self Stabilizing Systems (SSS 2005)*. LNCS, vol. 3764. Springer Verlag, Barcelona, Spain, 153–170.

KAYNAR, D. K., LYNCH, N., SEGALA, R., AND VAANDRAGER, F. 2006. *The Theory of Timed I/O Automata*, 1 ed. Morgan & Claypool Publishers.

KEIDAR, I. AND SHRAER, A. 2006. Timeliness, failure detectors, and consensus performance. In *Proceedings of the twenty-fifth annual ACM SIGACT-SIGOPS symposium on Principles of Distributed Computing (PODC'06)*. ACM Press, New York, NY, USA, 169–178.

KEIDAR, I. AND SHRAER, A. 2007. How to choose a timing model. In *Proceedings 37th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. 389–398.

KNUTH, D. E. 1966. Additional comments on a problem in concurrent programming control. *Communications of the ACM 9*, 5, 321–322.

KOTLA, R., ALVISI, L., DAHLIN, M., CLEMENT, A., AND WONG, E. L. 2007. Zyzzyva: speculative byzantine fault tolerance. In *SOSP*. 45–58.

KUTTEN, S. AND MASUZAWA, T. 2007. Output stability versus time till output. In *Distributed Computing, 21st International Symposium, DISC 2007, Lemesos, Cyprus, September 24-26, 2007, Proceedings*, A. Pelc, Ed. Lecture Notes in Computer Science, vol. 4731. Springer, 343–357.

LAMPORT, L. 1998. The part-time parliament. *ACM Transactions on Computer Systems 16*, 2 (May), 133–169.

LAMPORT, L. 2005. Fast Paxos. Tech. Rep. MSR-TR-2005-12, Microsoft Research.

LAMPORT, L. 2008. Computation and state machines. Unpublished note.

LAMPORT, L. 2009. The writings of Leslie Lamport, 155. How fast can eventual synchrony lead to consensus? `http://research.microsoft.com/en-us/um/people/lamport/pubs/pubs.html#PaxosGST`. [Online; March 26, 2009].

LAMPORT, L., SHOSTAK, R., AND PEASE, M. 1982. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems 4*, 3 (July), 382–401.

LAMPSON, B. 2001. The ABCD's of Paxos. In *PODC '01: Proceedings of the twentieth annual ACM symposium on Principles of distributed computing*. ACM, New York, NY, USA, 13.

LE LANN, G. 1977. Distributed systems—towards a formal approach. In *Information Processing 77*, B. Gilchrist, Ed. Proceedings of IFIP Congress, vol. 7. IFIP, North-Holland, Amsterdam, 155–160.

LINCOLN, P. AND RUSHBY, J. 1993. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Proceedings Fault Tolerant Computing Symposium 23*. Toulouse, France, 402–411.

LYNCH, N. 1996. *Distributed Algorithms*. Morgan Kaufman Publishers, Inc., San Francisco, USA.

MALKHI, D., OPREA, F., AND ZHOU, L. 2005. Ω meets paxos: Leader election and stability without eventual timely links. In *Proceedings of the 19th Symposium on Distributed Computing (DISC'05)*. LNCS, vol. 3724. Springer Verlag, Cracow, Poland, 199–213.

MARTIN, J.-P. AND ALVISI, L. 2006. Fast Byzantine consensus. *IEEE Transactions on Dependable and Secure Computing 3,* 3 (July), 202–215.

MATTERN, F. 1992. On the relativistic structure of logical time in distributed systems. In *Parallel and Distributed Algorithms*. Elsevier Science Publishers B.V, 215–226.

MERRITT, M. 1985. Notes on the dolev strong lower bound for Byzantine agreement. Unpublished note.

MEYER, F. J. AND PRADHAN, D. K. 1987. Consensus with dual failure modes. In *In Digest of Papers of the 17th International Symposium on Fault-Tolerant Computing*. Pittsburgh, 48–54.

MEYER, F. J. AND PRADHAN, D. K. 1991. Consensus with dual failure modes. *IEEE Transactions on Parallel and Distributed Systems 2,* 2 (Apr.), 214–222.

MOSER, H. 2009. Towards a real-time distributed computing model. *Theoretical Computer Science 410,* 6–7 (Feb), 629–659.

MOSER, H. AND SCHMID, U. 2006. Reconciling distributed computing models and real-time systems. In *Proceedings Work in Progress Session of the 27th IEEE Real-Time Systems Symposium (RTSS'06)*. Rio de Janeiro, Brazil, 73–76.

MULLENDER, S. 1993. *Distributed Systems*, 2nd ed. ed. ACM Press/Addison Wesley, New York.

NEIGER, G. AND TOUEG, S. 1990. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms 11*, 374–419.

NIST. 1992. The digital signature standard. *Communications of the ACM 35,* 7, 36–40.

PEASE, M., SHOSTAK, R., AND LAMPORT, L. 1980. Reaching agreement in the presence of faults. *Journal of the ACM 27,* 2 (April), 228–234.

PERRY, K. J. AND TOUEG, S. 1986. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering SE-12,* 3 (March), 477–482.

PINTER, S. S. AND SHINAHR, I. 1985. Distributed agreement in the presence of communication and process failures. In *Proceedings of the 14th IEEE Convention of Electrical & Electronics Engineers in Israel*.

RIVEST, R. L., SHAMIR, A., AND ADLEMAN, L. 1978. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM 21,* 12 (Feb.), 120–126.

ROBINSON, P. AND SCHMID, U. 2008. The Asynchronous Bounded-Cycle Model. In *Proceedings of the 10th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'08)*. Lecture Notes in Computer Science, vol. 5340. Springer Verlag, Detroit, USA, 246–262. (Best Paper Award).

RODRIGUES, R., CASTRO, M., AND LISKOV, B. 2001. BASE: Using abstraction to improve fault tolerance. In *Proceedings of the eighteenth ACM symposium on Operating systems principles (SOSP '01)*. ACM, New York, NY, USA, 15–28.

RUSHBY, J. 1994. A formally verified algorithm for clock sychronization under a hybrid fault model. In *Proceedings ACM Principles of Distributed Computing (PODC'94)*. Los Angeles, CA, 304–313.

SANTORO, N. 2007. *Design and Analysis of Distributed Algorithms*. Wiley Series on Parallel and Distributed Computing. John Wiley & Sons.

SANTORO, N. AND WIDMAYER, P. 1989. Time is not a healer. In *Proc. 6th Annual Symposium on Theor. Aspects of Computer Science (STACS'89)*. LNCS 349. Springer-Verlag, Paderborn, Germany, 304–313.

SANTORO, N. AND WIDMAYER, P. 1990. Distributed function evaluation in the presence of transmission faults. In *SIGAL International Symposium on Algorithms*. 358–367.

SAYEED, H. M., ABU-AMARA, M., AND ABU-AMARA, H. 1995. Optimal asynchronous agreement and leader election algorithm for complete networks with Byzantine faulty links. *Distributed Computing 9,* 3, 147–156.

SCHMID, U. 2000. Orthogonal accuracy clock synchronization. *Chicago Journal of Theoretical Computer Science 2000,* 3, 3–77.

SCHMID, U. AND SCHOSSMAIER, K. 2001. How to reconcile fault-tolerant interval intersection with the Lipschitz condition. *Distributed Computing 14,* 2 (May), 101 – 111.

SCHMID, U. AND WEISS, B. 2002. Synchronous Byzantine agreement under hybrid process and link failures. Tech. Rep. 183/1-124, Department of Automation, Technische Universität Wien. Nov. (replaces TR 183/1-110).

SCHMID, U., WEISS, B., AND KEIDAR, I. 2009. Impossibility results and lower bounds for consensus under link failures. *SIAM Journal on Computing 38,* 5, 1912–1951.

SCHMID, U., WEISS, B., AND RUSHBY, J. 2002. Formally verified Byzantine agreement in presence of link faults. In *22nd International Conference on Distributed Computing Systems (ICDCS'02)*. Vienna, Austria, 608–616.

SCHNEIDER, F. B. 1990. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys 22,* 4, 299–319.

SIU, H.-S., CHIN, Y.-H., AND YANG, W.-P. 1998a. Byzantine agreement in the presence of mixed faults on processors and links. *IEEE Transactions on Parallel and Distributed Systems 9,* 4 (Apr.), 335–345.

SIU, H.-S., CHIN, Y.-H., AND YANG, W.-P. 1998b. Byzantine agreement in the presence of mixed faults on processors and links. *IEEE Transactions on Parallel and Distributed Systems 9,* 4, 335–345.

SONG, Y. J. AND RENESSE, R. 2008. Bosco: One-step Byzantine asynchronous consensus. In *DISC '08: Proceedings of the 22nd international symposium on Distributed Computing*. Springer-Verlag, Berlin, Heidelberg, 438–450.

SRIKANTH, T. AND TOUEG, S. 1987a. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing 2*, 80–94.

SRIKANTH, T. K. AND TOUEG, S. 1987b. Optimal clock synchronization. *Journal of the ACM 34*, 3 (July), 626–645.

TEL, G. 1994. *Introduction to Distributed Algorithms*. Cambridge University Press.

THAMBIDURAI, P. M. AND PARK, Y. K. 1988. Interactive consistency with multiple failure modes. In *Proceedings 7th Symposium on Reliable Distributed Systems*. 93–100.

TOUEG, S., PERRY, K. J., AND SRIKANTH, T. K. 1987. Fast distributed agreement. *SIAM Journal on Computing 16*, 3, 445–457.

WALTER, C. J. AND SURI, N. 2002. The customizable fault/error model for dependable distributed systems. *Theoretical Computer Science 290*, 1223–1251.

WALTER, C. J., SURI, N., AND HUGUE, M. M. 1994. Continual on-line diagnosis of hybrid faults. In *Proceedings DCCA-4*.

WEISS, B. 2002. Authenticated consensus. Ph.D. thesis, Technische Universität Wien, Fakultät für Technische Naturwissenschaften und Informatik.

WEISS, B. AND SCHMID, U. 2001. Consensus with written messages under link faults. In *20th Symposium on Reliable Distributed Systems (SRDS'01)*. New Orleans, LA, USA, 194–197.

WIDDER, J. 2004. Distributed computing in the presence of bounded asynchrony. Ph.D. thesis, Vienna University of Technology, Fakultät für Informatik.

WIDDER, J., GRIDLING, G., WEISS, B., AND BLANQUART, J.-P. 2007. Synchronous consensus with mortal Byzantines. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'07)*. Edinburgh, UK, 102–111.

WIDDER, J. AND SCHMID, U. 2007. Booting clock synchronization in partially synchronous systems with hybrid process and link failures. *Distributed Computing 20*, 2 (Aug.), 115–140.

ZIELIŃSKI, P. 2004. Paxos at war. Tech. Rep. UCAM-CL-TR-593, University of Cambridge.