DISTRIBUTED PROTOCOLS ROBUST AGAINST MALICIOUS ATTACKS

BY

LUCIA DRAQUE PENSO

Sc.B. UNIVERSIDADE FEDERAL DO RIO DE JANEIRO, 1998

Sc.M. UNIVERSIDADE FEDERAL DO RIO DE JANEIRO, 2000

Sc.M. BROWN UNIVERSITY 2003

A DISSERTATION SUBMITTED IN PARTIAL FULFILLMENT OF

THE REQUIREMENTS FOR THE DEGREE OF

DOCTOR OF PHILOSOPHY

IN THE DEPARTMENT OF COMPUTER SCIENCE

AT BROWN UNIVERSITY

PROVIDENCE, RHODE ISLAND

MAY 2008

The dissertation by Lucia Draque Penso is accepted in its present form by the Department of Computer Science as satisfying the dissertation requirement for the Degree of Doctor of Philosophy.

Date ________________ ____

Maurice Herlihy, Advisor

Date ________________ ____

Felix Freiling, Advisor

Recommended to the Graduate Council

Date ________________ ________________________________

Anna Lysyanskaya, Reader

Approved by the Graduate Council

Date ________________ ____

Sheila Bonde, Dean of the Graduate School

Curriculum Vitae

Lucia Draque Penso, born on 16th October 1976 in Rio de Janeiro, a brazilian and portuguese citizen, received her Sc.M. degree from Brown University in 2003 and her Sc.M. and A.B. degrees from the Federal University of Rio de Janeiro in 2000 and 1998, respectively. Her Ph.D. advisors are Prof. Dr. Maurice Herlihy and Prof. Dr. Felix Freiling, the later having hosted her in Germany, initially at RWTH Aachen and later at the University of Manheimm. From 1998 to 2001, she also worked at BNDES, the Economical and Social Development National Bank, a brazilian government agency responsible for nurturing the economy and promoting progress in Brazil.

Her research focuses on theoretical and practical aspects of distributed computation and network security, with a special regard to wait-free agreement protocols. Nevertheless, she is also interested in problems stemming from the areas of economics, finance, cryptography and combinatorial optimization. She has a wide interest in theoretical problems and proof techniques which can be helpful in practical scenarios, and has already employed methods from probability theory, game theory and combinatorial topology, in order to develop optimal solutions or improve complexity analysis.

In particular, during the last years, she has worked extensively on methods, efficient algorithms and complexity theory for IT-security and fault-tolerant distributed systems. These have several practical applications, including, for instance, the secure and fast implementations of both data replication repositories and e-business software. Her research, resulting in several publications, has comprised the design, the complexity analysis and the implementation of deterministic and randomized, fault-tolerant and attack-resilient protocols in network environments containing miscellaneous dynamic distributed components.

Besides having taught classes at Brown University, at RWTH Aachen and at the University of Mannheim, she has co-advised the master dissertation two students from RWTH Aachen, namely, Andreas Tielmann and Marjan Ghajar-Azadanlou. She has also served as a referee for several conferences and magazines, including: DISC 2004, Euro-Par 2005, CC 2006, EDCC 2006, SRDS 2006, SSS 2006, SODA 2007, PODC 2007, SSS 2007, OPODIS 2007, IEEE Transactions on Software Engineering, Distributed Computing.

Abstract

In this thesis we investigate distinct techniques to avoid the impact of miscellaneous failures derived from hostile attacks in a message-passing distributed computing environment. Our failure palette assumes more benign faults as an appetizer and then moves on to more malign ones. To start with, we revisit the classical k-set agreement problem, where different distributed processors must all agree on up to k values previously proposed by them. We develop an optimal crash-resilient k-set agreement protocol, which tolerates the best possible number of crashes given (exact or close to) minimal synchrony features provided by limited-scope failure detectors in asynchronous systems. Tight bounds on the maximum number of crashes are achieved through combinatorial topology, and relation of failure detectors to timing assumptions, in settings such as stabilizing ones, is unfolded.

In the sequence, we broaden our failure repertoire to include message omissions performed by coprocessor hosts with high incentives to cheating. Here, coprocessors are tamper-proof and receive and send encrypted messages. This prevents arbitrary behavior of hosts, which may just omit incoming or outgoing messages or crash their own coprocessor. In this context, making use of secret shared coins, we derive randomized consensus (that is, 1-set agreement) protocols, optimal both in terms of time and resilience, and of very practical use for e-business. Deterministic versions and automatic transformations for failure detectors are also discussed.

Finally, we show how to boost threshold protocols in adversarial structure models where failures may be dependent and processors may behave badly in an arbitrary, byzantine way. In particular, we look at the problem of running any protocol that has an upper bound of $n > c\ t$, for any positive integer constant c, where n is the total number of processors and t is the maximum number of faults. We introduce an optimal byzantine-resilient protocol that enables simulation with less than n processors of any such threshold protocol in adversarial structure models. We also define equivalence classes using a particular set of key hierarchy properties.

Acknowledgments

First of all, I would like to thank Prof. Felix Freiling and Prof. Maurice Herlihy for the incredible technical, financial and emotional support throughout the several years invested in my Ph.D. thesis. Your advice and deep insight proved many times valuable and made me richer both as a professional and as a person. Your kindness, sensitivity, generosity, and smarts are something exceptional and a source of admiration. You helped me to unfold all my curiosity, ambition, desire for evolution and refinement, as well as interest in travelling to discover life and unveal distinct cultures and mindsets.

Secondly, I would like to thank Prof. Anna Lysyanskaya for joining my thesis committee - she is definitely a role model for successful and intelligent women in science who do not want to lose charm, elegance, tenderness and vivacity. I am also grateful to the staff of the University of Mannheim (Sabine Braak and Jürgen Jaap) and Brown University (Ayanna Belton, Eugenia deGouveia, Kathy Kirman, Dorinda Moulton, Patrick Paul and Max Salvas) for the help, as well as to the distributed computing groups of the University of Paris 7 (Carole Delporte-Gallet, Hugues Fauconnier and Andreas Tielmann), the University of the Basque Country at San Sebastian (Roberto Cortiñas, Alberto LaFuente, Mikel Larrea, Iratxe Soraluze Arriola), the University of Vienna (Martin Biely, Martin Hutle and Josef Widder), the University of Mannheim (Michael Becher, Zinaida Benenson, Thorsten Holz and Martin Mink), the University of Texas at Dallas (Neeraj Mittal) and Yahoo! at Barcelona (Flávio Junqueira) for the fruitful discussions and the warm and cordial hospitality.

Clearly, I cannot pass without mentioning the substantial contribution of my family, who has constantly loved and encouraged me, in special Dieter Rautenbach, my parents Natércia and Antônio, and my older brother Antônio Luís. You are a permanent motivation for me to carry on pursuing my goals in life.

Last, I would like to thank all wonderful souls who made me laugh and enjoy myself all along the way, proportionating treasurable moments of happiness and distraction, keeping the good humor, making my life brighter, and fostering the fun-loving and light-hearted side of my personality, which I have always been very proud of.

Contents

1 Introductory Overview **1**
1.1 Motivation . . . 1
1.2 Contributions . . . 2
1.3 Thesis Outline . . . 4

2 Distributed System Models **5**
2.1 Distributed Computing . . . 5
2.2 Failures . . . 6
2.2.1 Failure Models for Processors or Automata . . . 6
2.2.2 Failure Models for Communication Channels . . . 7
2.3 Timing Models . . . 8
2.3.1 Synchronous Systems . . . 8
2.3.2 Partially Synchronous Systems . . . 9
2.3.3 Asynchronous Systems . . . 9
2.4 K-Set Agreement . . . 9
2.5 Failure Detectors . . . 9
2.6 Adversary Model . . . 10
2.7 Summary . . . 10

3 A Crash Robust Protocol and Its Lower Bound **11**
3.1 Motivation . . . 11
3.2 Related Work . . . 13
3.3 Topological Model . . . 14
3.4 Models of Computation . . . 16
3.5 Perpetual Weak (x, q)-Accuracy . . . 17
3.5.1 No Clusters . . . 17
3.5.2 Single Cluster . . . 18
3.5.3 Multiple Clusters . . . 20
3.6 Eventual Weak (x, q)-Accuracy . . . 23

4 Robust Crash Transformations **27**
4.1 Motivation . . . 27
4.1.1 Contribution . . . 28
4.2 System Models . . . 30
4.2.1 Common Definitions . . . 30
4.2.2 The Failure Detector Model . . . 31
4.2.3 The Partially Synchronous Model . . . 31
4.2.4 The Eventually Synchronous Model . . . 32
4.2.5 The Round Model . . . 33

4.2.6 Algorithm Transformations . . . 33
4.3 Equivalence of Solvability . . . 35
4.3.1 Possibility of **Async+$\mathcal{G}$⇀ParSync** . . . 35
4.3.2 Possibility of **ParSync⇀◇Sync** . . . 35
4.3.3 Possibility of **◇Sync⇀Round** . . . 37
4.3.4 Possibility of **Round⇀Async+$\mathcal{G}$** . . . 39
4.4 Efficiency of Transformations . . . 40
4.4.1 Lower Bounds . . . 40
4.4.2 Upper Bound on **Async+$\mathcal{G}$⇀◇Sync** . . . 42
4.5 Discussion . . . 43

5 Randomized Message Omission Robust Protocols 45
5.1 Motivation . . . 45
5.2 Model of Computation . . . 47
5.3 Fair Exchange as Consensus . . . 48
5.4 Related Work . . . 49
5.5 Optimal Protocol for Send Omissions . . . 49
5.6 Optimal Protocol for Send and Receive Omissions . . . 52

6 Deterministic Message Omission Robust Protocols 55
6.1 Motivation . . . 55
6.2 System Model and Architecture . . . 59
6.2.1 Untrusted and Trusted System . . . 59
6.2.2 Timing Assumptions . . . 60
6.2.3 Failure Assumptions . . . 60
6.2.4 Classes of Processes in the Trusted System . . . 60
6.2.5 The TrustedPals Architecture . . . 61
6.3 Failure Detection in TrustedPals . . . 62
6.4 $\Diamond\mathcal{P}$-based Consensus in TrustedPals . . . 67
6.5 Integrating Failure Detection and Consensus Securely . . . 70
6.6 Security Evaluation . . . 72
6.6.1 Analysis . . . 72
6.6.2 Summary . . . 76

7 Message Omission Robust Weakest Failure Detectors 77
7.1 Motivation . . . 77
7.2 Model . . . 79
7.3 From Crash-Stop to Permanent Omission . . . 81
7.3.1 The General Transformation . . . 82
7.3.2 The Transformation for $n > 2f$ . . . 83
7.4 Results . . . 85

8 Byzantine Robust Protocols 93
8.1 Motivation . . . 94
8.2 System model . . . 95
8.3 Replication with cores and survivor sets . . . 97
8.4 Constructing protocols . . . 98
8.5 Equivalence classes of adversarial structures . . . 102

9 Summary and Future Work 103

List of Figures

3.1 The TWA_q Protocol 23
3.2 k-set agreement protocol for $S_{x,q}$ 24
3.3 k-set agreement protocol for $\diamond S_{x,q}$ 25

4.1 Transforming Async+$\mathcal{G}$ algorithms to ParSync algorithms. 36
4.2 Transforming $\diamond$**Sync** algorithms to **Round** algorithms (Algorithm 2) and Transforming **Round** algorithms to **Async**+$\mathcal{G}$ algorithms (Algorithm 3). . 38
4.3 Relations of models with pointers to sections here; an arrow from model M_1 to model M_2 indicates that a result on algorithm transformations from model M_1 to model M_2 can be found here. Solid lines indicate upper bounds, dotted lines indicate lower bounds. $B|D$ means that a transformation exists which is B-bounded and has D-bounded shift. We use *nep* to mark non-efficiency-preserving transformations. 43

5.1 Untrusted parties and security modules. 46
5.2 The untrusted system and the trusted subsystem. 47
5.3 Uniform consensus for send message omissions and process crashes. 50
5.4 Uniform consensus for send and receive message omissions and process crashes. 53

6.1 Processes with tamper proof security modules. 56
6.2 The untrusted and trusted system. 59
6.3 Examples for classes of processes. 61
6.4 The architecture of our system. 62
6.5 $\diamond\mathcal{P}$ in the omission model: main algorithm. 64
6.6 $\diamond\mathcal{P}$ in the omission model: procedure update_In_Out_Connected_lists(). . . 65
6.7 $\diamond\mathcal{P}$ in the omission model: procedure deliver_next_message(). 65
6.8 Solving consensus in the omission model using $\diamond\mathcal{P}$: main algorithm. 68
6.9 Solving consensus in the omission model using $\diamond\mathcal{P}$: adopting the decision. . 68
6.10 Smartcard with scrambler. 70
6.11 Example of scrambler's function. 71
6.12 Part of scrambler's function in pseudocode. 72
6.13 Attack tree for the threat of safety properties. 73
6.14 Attack tree for the threat of liveness properties. 74
6.15 Attack tree for the threat of information flow properties. 75

7.1 Additional Communication Layers 83
7.2 The Three Way Handshake Algorithm for Process p_i. 84
7.3 The Relaying Algorithm for Process p_i. 84
7.4 Transformations of the Consensus Problem 85
7.5 The Two Way Handshake Algorithm for Process p_i. 86

7.6 Construction of H' 87

9.1 Channel directed from a correct process q to a malicious process p. 105
9.2 Channel directed from a malicious process q to a correct process p. 105

Chapter 1

Introductory Overview

1.1 Motivation

In distributed systems all safety, liveness and security aspects of an application play a crucial role. Safety guarantees that execution is correctly performed. Liveness determines what at some point must come to an end. Security prevents leak of private information and misusage by non-authenticated users [16, 87, 105]. Moreover, as the size of a distributed system grows, so does the number of its components and the probability that some might fail or be manipulated — for instance, due to ambition (that is, advantage search for oneself) or envy (that is, simple pleasure of damaging others) [34, 69, 86]. Depending on the nature of such a fault or attack, it may happen in either independent or correlated fashions — for example, when there is a collusion of interests, sabotage devised by a group may in fact be coordinated to maximize individual profit of its elements [5, 70, 77, 73]. Therefore, designers must always incorporate from the very start fault-tolerance and security mechanisms into distributed protocols when developing a new reliable distributed system, given the spectrum of possible failures and moves from an adversary [22, 62, 81, 97, 98, 107, 110, 116].

In this thesis we mainly focus on the feasibility of solving agreement problems [31] efficiently and securely in the presence of faults which vary from crash to byzantine [80] (that is, arbitrary or malicious behavior) in distributed systems. Furthermore, for message omission failures, randomized protocols here derived [54]have direct application in electronic commerce. In fact, such agreement problems are essencial for a number of purposes, including plane and robot control [106, 26], group-communication [13], data replication [63], and e-voting [57] - or more generally, secure multi-party computation [17, 53].

More precisely, in the k-set agreement problem, introduced by Chaudhuri [31, 32, 64, 93], each process in a group starts with a private input value, communicates with the others, and then halts after choosing a private output value. Each process is required to choose some process's input, and at most k distinct values may be chosen. Note that, depending on the failure model, such a definition might become slightly changed. In particular, 1-set agreement is simply named consensus. As said, solving k-set agreement is key to many other problems.

Moreover, all protocols and automatic transformations considered here involve at least one out of three types of faults: process crash [82], (send or/and receive) message omission [102], or byzantine (arbitrary) [80]. In special, message omission models are justified by the presence of processes equipped with trusted coprocessors such as smartcards, which may be used to relate security problems to safety problems and conversely [17, 53]. Note that arbitrary failures comprise message omission failures which comprise crash failures.

Faults may be derived from attackers, which is very likely given that practically every system is today not only distributed but in fact connected to wide area networks or the Internet.

Hence, despite such a scenario these distributed networks must be protected from faulty behavior in order to guarantee user reliability and dependability. In short, whenever necessary, there should be a way to guarantee the authenticity of code lines, authentication of communicating parties, and protection of messages against tampering and deletion, in order to provide both protocol confidentiality and correct functioning.

Thus, to ensure correct execution of a protocol and to inhibit a potential adversary to corrupt or destroy its output or to obtain information, here we incorporate (safety, liveness and security) mechanisms into a protocol (or an automatic transformation on it) which can prevent adversaries of performing a successful attack. In the following chapters, given specific problems in a distributed environment, we first identify security issues and potential vulnerabilities of the distributed system which could have an impact during a distributed protocol execution, so that we can develop strategies to overcome the possible different types of attacks.

1.2 Contributions

Here, we collect a series of correlated papers published in some of the most competitive and respectable conferences and journals in the area, following an american tradition at computer science departments of focusing on publication acceptance at prestigious venues rather than practice of storytelling skills. More precisely, in this thesis:

- We give a new optimal (in terms of resilience) deterministic consensus protocol and its related lower bounds in an asynchronous environment with a special cluster-based failure detection. Joint work with Maurice Herlihy, *"Tight Bounds for k-Set Agreement with Limited-Scope Failure Detectors"*, published in *Distributed Computing, 18(2), pages 157—166, July 2005* [64].

- We show, through protocol automatic transformations, that distinct computational models with stabilizing properties in the crash model are equivalent regarding solvability, and analyze efficiency issues. Such models include e.g. the partially synchronous model [48], where eventually the distributed system obeys bounds on computing speeds and message delays, or the asynchronous distributed system model augmented with unreliable failure detectors [28], where eventually failure detectors do not make mistakes. Joint work with Martin Biely, Martin Hutle and Josef Widder, *"Relating Stabilizing Timing Assumptions to Stabilizing Failure Detectors Regarding Solvability and Efficiency"*, published in *Proceedings of the Ninth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2007), pages 4—20, November 2007, Paris, France*, [18].

- We give a novel optimal (in terms of resilience and time) randomized consensus protocol in a message omission synchronous setting with trusted coprocessors, useful for e-business. Joint work with Felix Freiling and Maurice Herlihy, *"Optimal Randomized Fair Exchange with Secret Shared Coins"*, published in *Proceedings of the Ninth International Conference on Principles of Distributed Systems (OPODIS 2005), pages 61—72, December 2005, Pisa, Italy* [54].

- We investigate the feasibility of providing a deterministic, efficient and secure solution to consensus in the presence of partial synchrony with a certain failure detection in a message omission asynchronous setting with trusted coprocessors. Joint work with Roberto Cortiñas, Felix Freiling, Marjan Ghajar-Azadanlou, Alberto Lafuente, Mikel Larrea and Iratxe Arriola Soraluze, *"Secure Failure Detection in TrustedPals"*, published in *Proceedings of the Ninth International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2007), pages 173—188, November 2007, Paris, France*, [36].

- We give a new failure detection automatic transformation from the crash model to the message omission model, which is weakest failure detection preserving. Joint work with Carole Deporte-Gallet, Hugues Fauconnier, Felix Freiling and Andreas Tielmann, *"From Crash-Stop to Permanent Omission: Automatic Transformation and Weak Failure Detectors"*, published in *Proceedings of the Twenty-First International Symposium on Distributed Computing (DISC 2007), pages 165—178, September, 2007, Lemesos, Cyprus*, [41].

- We give a novel protocol automatic transformation that enables simulation with less processors for a class of threshold protocols which also comprises problems other than agreement ones, and which is resilient to byzantine failures. Equivalence classes in terms of solvability are also defined. Joint (unpublished) work with Maurice Herlihy, Flávio Junqueira and Keith Marzullo.

Other work done on the way to the Ph.D. degree which is not comprised in this thesis can be found in the following publications:

- *"Efficient Reduction for Wait-Free Termination Detection in a Crash-Prone Distributed System"*, joint work with Neeraj Mittal, Felix Freiling and Subbarayan Venkatesan, published in *Proceedings of the Nineteenth International Conference on Distributed Computing (DISC 2005), pages 93—107, September, 2005, Cracow, Poland*, [90].

- *"TrustedPals: Secure Multiparty Computation Implemented with Smartcards"*, joint work with Milan Fort, Felix Freiling, Zinaida Benenson and Dogan Kesdogan, published in *11th European Symposium on Research in Computer Security (ESORICS 2006), pages 306—314, September, 2006, Hamburg, Germany*, [53].

- *"Safety, Liveness, and Information Flow: Dependability Revisited"*, joint work with Zinaida Benenson, Felix Freiling, Dogan Kesdogan and Thorsten Holz, published in *Proceedings of the 4th ARCS International Workshop on Information Security Applications, pages 56—65, March, 2006, Frankfurt am Main, Germany*, [16].

1.3 Thesis Outline

The upcoming chapters are organized as follows. Chapter 2 presents an overview of the basic concepts in distributed computing and related areas relevant to this thesis. However, due to the large variety of scenarios, note that each distinct chapter precisely describes the system model to be considered within it. For clarity purposes, the same happens with the related work.

Chapter 3 introduces an optimal crash robust protocol for the k-set agreement problem and its related lower bounds in a message-passing asynchronous distributed system under special assumptions [64]. Another paper by us, which plays a role in the crash model but is not included in this thesis, presents a crash robust protocol for wait-free termination detection [90]. Chapter 4 presents relation between different computational models regarding solvability, still in the crash model, and analyzes efficiency issues [18].

In Chapter 5, optimal randomized consensus protocols in a message omission synchronous setting with trusted coprocessors are shown [54]. Another paper by us, not included in this thesis, which implements synchronous deterministic solutions prior to ones in Chapter 5, in a system environment called TrustedPals to be used in Chapter 6 under an asynchronous perspective, can be found in [53]. Moreover, related discussions by us on dependability issues can be found in [16]. Chapter 6 investigates the feasibility of providing a deterministic, efficient and secure solution to consensus in the presence of partial synchrony with a certain failure detection in a message omission asynchronous setting with trusted coprocessors provided by TrustedPals [36]. In Chapter 7, a weakest failure detection preserving automatic transformation from the crash model to the message omission model is given [41].

In Chapter 8, a novel byzantine resilient protocol automatic transformation that enables simulation with less processors for a class of threshold protocols is displayed. Finally, Chapter 9 concludes with a summary and a handful of suggestions for future work.

Chapter 2

Distributed System Models

Before presenting the main results which comprise this thesis, we quickly review some of the basic concepts and issues related to distributed computing [82, 60]. Hence, this chapter gives an overview of the formal models used in the following chapters. Nonetheless, note that each chapter precisely describes model(s) there assumed. In Section 2.1 a brief description of a distributed system components is given. Then, in Section 2.2 a list of possible failures is shown, while in Section some general timing models are introduced. Finally, a short overview, on important components of distributed systems in Section 2.5 and possible considerations of adversaries in Section2.6, is given.

2.1 Distributed Computing

A *distributed system* [82] is modeled as a directed graph of n vertices, which denote processors, while channels are denoted by directed edges. Each processor contains at least one process, but may contain several. Moreover, each process has an outgoing message buffer and an incoming message buffer to/from every other processor.

I/O automata are used to model processes and channels. Formally, an I/O automaton consists of: a set of *start states*, a set of *states*, *final states*, *actions*, and a *transition function.* Besides, each automaton has *external actions*, *internal actions*, *input actions* and *output actions.* Note that while internal and output actions depend on the automaton, external and input actions depend on the environment. For instance, an internal action may be querying an oracle, such as querying a failure detector, a device explained later in one of the next sections, and an external action may be another automaton crash. Furthermore, outputs may be of the form $send_{i,j}(m)$, where m is a message sent from automaton i to another automaton j, and inputs may be of the form $receive_{i,j}(m)$, where m is a message received at automaton i from another automaton j. Note that when automaton a_i sends a message m to automaton a_j, a_i inserts m into its outgoing message buffer. From there m is transported over the channel to a_j's incoming message buffer, from where it is then removed and received by a_j.

A *run* of an I/O automaton is either a finite sequence of steps, $s_0, c_1, s_1, \ldots, c_r, s_r$, or an infinite sequence $s_0, c_1, s_1, \ldots, c_r, s_r, ...$ of alternating states and actions, where s_0 is an initial state, c_i is an action, s_i is a state, and each triple (s_{i-1}, c_i, s_i) is a transition of for every $i \geq 1$. Each transition is only modified by, and only modifies, the *local state* of a single automaton. c_i is said to be *enabled* in s_{i-1} when (s_{i-1}, c_i, s_i) is a possible step. Automata are *input-enabled* if every input action is enabled in every state.

An *execution* is a sequence of global states of the system, where each *global state* comprises the state of each automaton and the collection of messages in transit. In this scenario, the *events* of an automaton are the local transitions. A *behavior* of an execution is the consists of external actions.

A *problem* to be solved by an automaton is a set of sequences of external actions. In a distributed system, a *distributed protocol* for a collection of processes is a collection of local protocols, one for each process, each process represented by an automaton.

A *problem specification* is a set of possible behaviors and an automaton solves the specification if each of its behaviors is contained in this set. A *property* is a subset of a problem specification. There are three main types of properties in distributed computing:

- *Safety*: A safety property states that something will never happen. An example of a safety property would be value agreement, i.e. no two processes can decide on different values.
- *Liveness*: A liveness property states that eventually something will happen. An example of a liveness property would be value decision, i.e. at some point a decision on a value is taken.
- *Information flow*: An information flow property states which information is can be learned and who is authorized to learn it. A property here is a set of set of executions. An example is that of confidential channels, i.e. the fact that the content of a message m sent remains confidential to other processes [87].

2.2 Failures

Failure models characterize the possible scope of faults within a distributed computing environment [82, 60]. Here, a component in a distributed system such as a process or a channel is said to be *correct* if it does not perform any failures at all or it does not operate maliciously, otherwise it is *faulty.*

Failures can be classified as *transient* or *permanent.* A transient fault will eventually vanish, whereas a permanent one will remain. A protocol coping with t failures is said to be *t-resilient*, and if $t = n - 1$, it is said to be $wait - free$.

2.2.1 Failure Models for Processors or Automata

Now we introduce various forms of processor or automaton failures [82]. Note that if a processor is contaminated by a kind of failure, so are its automata.

Crash

A *crash failure* occurs when a processors' automaton halts its protocol and all its external communication, never recovering again. A typical reason for a crash is an operating system that comes to a halt.

Message Omission

In a *message omission failure*, a processor's automaton may crash or experience a send or receive omission, i.e. it does not send a message it is supposed send to its outgoing buffer or does not receive a message it is supposed to receive in its incoming buffer, according to its protocol.

Byzantine

A *byzantine failure* is an arbitrary fault that occurs during the execution of a processor's automaton protocol, be it by purpose or a random event, representing unexpected hardware failures, network congestions and disconnections, and malicious attacks.

2.2.2 Failure Models for Communication Channels

Here we present distinct channels, which are supposed to carry and deliver messages from outgoing buffers to incoming buffers. Each type of channel is subject to different faults, and are listed in the following from the weakest to the strongest in terms of reliability [60].

Fair-loss Channels

A *fair-loss channel* does not ensure a reliable connection, since messages can be lost in transit. Fair-loss channels pass on an infinite number of messages if they received an infinite number of messages. Fair-loss channels satisfy the following properties:

- *Fair-Loss*: If a process p sends an infinite number of messages m to process q, then q receives an infinite number of messages from p.
- *Finite Duplication*: If a message m is sent a finite number of times by a process p to a process q, then m cannot be delivered an infinite number of times by q.
- *No Creation*: If a message m is delivered by some process q, then m was previously sent by some process p, and p and q are both correct.

Best-effort Channels

Best-effort channels, do not provide any guarantees that the message is delivered if the sender fails. A best-effort channel satisfies the following properties:

- *Best-effort*: If a process p sends a message m to process q, and none of the both processes p and q has failed, then q eventually receives m.
- *No Duplication*: No message is delivered by a process more than once.
- *No Creation*: If a message m is delivered by some process q, then m was previously sent by some process p, and p and q are both correct.

Stubborn Channels

Stubborn channels establish a bridge between best-effort channels and reliable channels which are introduced in the next subsection. A stubborn channel satisfies the following properties:

- *Stubborn*: If a process p sends a message m to a process q and p does not fail, then q eventually delivers m an infinite number of times.
- *No Creation*: If a message m is delivered by some process q, then m was previously sent by some process p, and p and q are both correct.

Reliable Channels

A *reliable channel* can be characterized as a channel where all messages which are sent are received, even if the sending process crashes after having sent the message, but provided that the receiver process does not crash. It is a useful abstraction when designing and proving the correctness of distributed algorithms and it has stronger semantics than best-effort channels and stubborn channels. A reliable channel can be defined as a channel that satisfies the following properties:

- *No Loss*: If a process p sends a message m to process q, and p and q do not fail, then q eventually receives m.
- *No Duplication*: No message is delivered by a process more than once.
- *No Creation*: If a message m is delivered by some process q, then m was previously sent by some process p, and p and q are both correct.

Secure Channels

A *secure channel* is a reliable channel from which an adversary does not have the ability to delete, modify, insert, or read a message. Protecting messages against eavesdropping is done by ensuring confidentiality. Protecting against modification and insertion is done by protocols for mutual authentication and message integrity. Thus, a secure channel satisfies the following properties:

- *No Loss*: If a process p sends a message m to process q, and p and q do not fail, then q eventually receives m.
- *No Duplication*: No message is delivered by a process more than once.
- *No Creation*: If a message m is delivered by some process q, then m was previously sent by some process p, and p and q are both correct.
- *Confidentality*: If a process p sends a message m to process q, and p and q do not fail, then the content of the message m is not accessible to unauthorized parties.

2.3 Timing Models

An essential aspect of defining a distributed systems model is the automata and channel behavior in respect to time, which in the context of this thesis may happen in three distinct ways.

2.3.1 Synchronous Systems

In a *synchronous* system one has exact upper bounds on processing and communication delays. Clearly, if bounds exist and are known, it is easier for an automaton to detect the failure of others. A system is synchronous if it satisfies the following properties:

- There exists a known upper bound on message delay.
- There exist known upper bounds on the time necessary to execute a protocol step.

2.3.2 Partially Synchronous Systems

An alternative on weakening the assumptions on explicit timing bounds is the *partial synchrony* model. Here, there are three possibilities of partial synchrony:

- message delay and processing speeds may be bounded, but the bound may be unknown, or
- bound may be known but holds only eventually, or
- bound may be unknown and holds eventually.

2.3.3 Asynchronous Systems

Last, in an *asynchronous* system there are no assumptions about the relative speeds of the automata or their channels. It is assumed that components take steps in an arbitrary order, at arbitrary relative speeds, in a way similar to real distributed systems such as the Internet.

2.4 K-Set Agreement

Agreement problems represent an essential building block of the whole distributed computing area, as noted in our first chapter. The most general is the k-set agreement problem [31], where each process in a group starts with a private input value, communicates with the others, and then halts after choosing a private output value. Each automaton is required to satisfy the following properties:

- *Termination*: Every correct automaton eventually decides on some value.
- *Validity*: The decided value must have been proposed by some automaton.
- *Agreement*: The decided value differs from at most $k - 1$ distinct values decided by other automata.

This definition may be modified according to the failure model. Besides, when $k = 1$, this problem is more commonly named by consensus.

2.5 Failure Detectors

To circumvent asynchronicity issues such as problem solving impossibilities, an asynchronous distributed system component called *failure detector* is used [74]. More precisely, a *failure detector* is an abstract device provided to a system to enable automata to know more about its external distributed environment. In short, each automaton in the system is associated with a local failure detector, which it can query at any time, in order to get a list of automata or channels that the failure detector suspects to have failed at that moment.

Usually a failure detector can be specified by a completeness and an accuracy property. Common completeness properties used to define failure detectors in environments prone to crash failures:

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

- *Weak completeness*: Eventually every automaton that crashes is permanently suspected by some correct automaton.

Besides, four common accuracy properties also used to define failure detectors are:

- *Strong accuracy*: No automaton is suspected before it crashes.
- *Weak accuracy*: Some correct automaton is never suspected.
- *Eventual strong accuracy*: There is a time after which correct automata are not suspected by any correct automaton.
- *Eventual weak accuracy*: There is a time after which some correct automaton is never suspected by any correct automaton.

Miscellaneous failure detectors are defined within this thesis, including ones which deal with more challenging types of failures. Note that each chapter provides its own definition of failure detection according to its failure model and requirements stemming from problem under investigation.

2.6 Adversary Model

We say a processor or automaton is malicious if any protocol running inside is not executed as it is supposed to. Otherwise the processor or automaton is correct.

There are two different assumptions about the computational power of an adversary, i.e. the quantity of resources the adversary can afford in order to succeed [70]. We distinguish between:

- *Bounded Adversary*: A bounded adversary can only use a restricted amount of time and space.
- *Unbounded Adversary*: An unbounded adversary can use as much time and space as it needs.

In this thesis, it is assumed that adversaries are computationally bounded. Furthermore, it is assumed that adversaries may be defined using fail-prone systems [74], which define maximal sets of automatons that can fail at the same time, a way to define the scope and power of an adversary's influence. A more precise description of which type of adversary we are dealing with is given in each chapter, respectively.

2.7 Summary

In this chapter, we briefly introduced a variety of basic notions and assumptions from the distributed computing area to be used in the next chapters.

Chapter 3

A Crash Robust Protocol and Its Lower Bound

We now start with our presentation of fault-resilient distributed protocols, by considering more simple failures such as crash ones. Here, we introduce a new optimal crash robust protocol for the k-set agreement problem in a message-passing asynchronous distributed system under special assumptions [64]. Furthermore, we prove lower bounds related to it, that is, the one which proves its optimality and another one which proves the optimality of a very close related crash-resilient distributed protocol. Note that the concept of optimal refers to the maximum number of faults which may be tolerated under the considered model. In the following sections, we clarify the contributions (Section 3.1), describe the related work (Section 3.2), give details on the assumed topological and computation models (Section 3.3 and Section 3.4), prove a lower bound for a related model (Section 3.5), and finally introduce our novel optimal crash robust protocol and prove its lower bound (Section 3.6).

3.1 Motivation

As a reminder, in the k-set agreement problem [31], each process in a group starts with a private input value, communicates with the others, and then halts after choosing a private output value. Each process is required to choose some process's input, and at most k distinct values may be chosen.

We consider this problem in an asynchronous message-passing system of $n+1$ processes, of which at most f may fail by halting. [1] Each process is equipped with a failure detector [27, 28], an unreliable oracle that continually provides the process with a list of processes *suspected* of having failed. As said before, a failure detector is a mathematical abstraction that models time-out, heartbeat, and related techniques used by real systems to detect failures. A process that is not suspected is *trusted.* A *correct* process is one that never fails, and a *non-faulty* process at a particular time (sometimes implicit) is one that has not failed yet.

Limited-scope failure detectors [94, 117] formally capture the idea that a process can typically detect some failures more accurately than others. For example, if processes send one another periodic heartbeat messages, then a process may detect failures reliably on the same local-area network, but less reliably over a wide-area network.

[1]When comparing our formulas to those of Moustéfaoui and Raynal [94], be aware that they assume n processes in the system, while we assume $n+1$, which simplifies topological calculations.

Mostéfaoui and Raynal [94] propose a model in which there is a single cluster of processes, containing at least one *correct* (that is, not failed) process that is never erroneously suspected by any process in that cluster. Here, we extend that model in a natural way to admit multiple clusters. Each cluster includes at least one correct process that is never suspected by any process in that cluster, either from the very beginning or eventually, i.e., after some point in time. A model with q clusters corresponds to a network composed of q local area networks.

As we will show, the circumstances under which k-set agreement can be solved in this model are determined by the values of k, of q (the number of clusters), and x, the combined size of the $\min\{k, q\}$ largest clusters (the k, if $q \geq k$, or the q largest, if $q < k$).

As described in Chapter 2, a failure detector class is characterized by a *completeness* property and an *accuracy* property. The limited-scope failure detectors considered here satisfy

- **Strong Completeness:** Any process that crashes is eventually permanently suspected by every correct process.

Informally, if a process really crashes, then sooner or later, every failure detector will detect that something is wrong.

We consider two alternative accuracy properties.

- **Perpetual Weak (x, q)-Accuracy:** Some correct process in each cluster is never suspected by any process in that cluster.
- **Eventual Weak (x, q)-Accuracy:** Eventually, there is a time after which some correct process in each cluster is never suspected by any process in that cluster.

We focus on two failure detector classes in this paper:

- $S_{x,q}$ satisfies strong completeness and perpetual weak (x, q)-accuracy.
- $\diamond S_{x,q}$ satisfies strong completeness and eventual weak (x, q)-accuracy.

In a system of $n + 1$ processes, the well-known failure detector S introduced by Chandra and Toueg [28] is just $S_{n+1,1}$, and $\diamond S$ is $\diamond S_{n+1,1}$. The *limited-scope* failure detectors considered by Mostéfaoui and Raynal [94], S_x and $\diamond S_x$ are $S_{x,1}$ and $\diamond S_{x,1}$. Note that, by definition, both $S_{x,q}$ and $\diamond S$ are at least as strong as $\diamond S_{x,q}$.

We make the following contributions.

Lower Bounds We give the first lower bounds for k-set agreement protocols employing failure detector classes $S_{x,q}$ and $\diamond S_{x,q}$. For the perpetually-accurate class $S_{x,q}$, we show that no k-set agreement protocol is possible if

$$f \geq \begin{cases} k + x - q & q \leq k \\ x & \text{otherwise.} \end{cases}$$

In the special case where there is only one cluster, our lower bound implies that the elegant *TWA-based* protocol of Mostéfaoui and Raynal [94] is optimal, confirming their conjecture.

For the eventually-accurate class $\diamond S_{x,q}$, we show that no $(n+1)$-process k-set agreement protocol is possible if

$$f \geq \begin{cases} \min(\frac{n+1}{2}, k + x - q) & q \leq k \\ \min(\frac{n+1}{2}, x) & \text{otherwise.} \end{cases}$$

In the special case where there is only one cluster, there is a gap between our lower bound and the algorithm proposed by Mostéfaoui and Raynal. We close this gap with a new algorithm, described below.

Our proof employs concepts and methods adapted from elementary Combinatorial Topology [66, 68]. These methods have been successful in other models, but this is the first time such methods have been applied to failure detectors.

Note that, our proof implies that given t, S_{x^*,q^*} is weakest failure detector for $k = t - x^* + q^* + 1$ among class of $S_{x,q}$ failure detectors, for all possible x and q.

Upper Bounds For perpetually-accurate failure detectors (class $S_{x,q}$), we give a simple but non-trivial generalization of the Mostéfaoui and Raynal algorithm. The new algorithm encompasses multiple clusters, and it is optimal because it matches our lower bound.

For eventually-accurate failure detectors (class $\diamond S_{x,q}$), we give a novel protocol that matches our lower bound. In the special case where there is only one cluster, our protocol improves on the corresponding protocol of Mostéfaoui and Raynal, disproving their conjecture that their protocol is optimal.

Our protocol has an unexpectedly simple structure: It alternates the perpetually-accurate protocol with a novel *convergence detection* protocol that halts when it detects that an earlier iteration of the perpetually-accurate protocol has succeeded.

3.2 Related Work

We build on pioneering work of Mostéfaoui and Raynal [94]. We show that their TWA-based protocol for S_x is optimal if there is only one cluster, and we improve their protocol for $\diamond S_x$ from

$$f < max(k, max_{1 \leq \alpha \leq k}(min(n + 1 - \alpha \left\lfloor \frac{n+1}{\alpha+1} \right\rfloor, \alpha + x - 1)))$$

to an optimal

$$f < min(\frac{n+1}{2}, k + x - 1).$$

Moreover, we generalize their protocol to q clusters, instead of one, and show that this generalized protocol is optimal for $S_{x,q}$. We also show that our new protocol is optimal for $\diamond S_{x,q}$.

Anceaume et al. [6] give a non-optimal multi-module k-set agreement protocol for $\diamond S_{x,1}$ that, besides tolerating $f < \frac{n+k-1}{2}$ failures when $x > f$, also becomes communication-efficient if at least one of $k - 1$ pre-determined processes does not fail.

Borowsky and Gafni [21], Herlihy and Shavit [68], and Saks and Zaharoglou [108] showed there is no wait-free protocol for k-set agreement in asynchronous message-passing or read/write memory models. Chaudhuri, Herlihy, Lynch, and Tuttle [32], Herlihy, Rajsbaum, and Tuttle [66, 67] derive lower bounds on round complexity for the synchronous fail-stop message-passing mode. Many of these proofs rely, directly or indirectly on mechanisms and techniques adapted from Combinatorial Topology.

Failure detectors [27, 28] have received an enormous amount of attention, most of which has focused on solving the consensus problem. Yang, Neiger, and Gafni [117], and Mostéfaoui and Raynal [94] have proposed k-set agreement protocols for models that encompass limited-scope failure detectors, but we are unaware of any prior lower bounds for these models.

Gafni [56] introduces the notion of *round-by-round* failure detectors to give a number of novel reductions between models.

Attiya and Avidor [8] and Mostéfaoui et al. [91] have investigated the related problem of solving k-set agreement when inputs are restricted.

Our new algorithm for eventual weak accuracy failure detectors has a style similar to the *k-converge algorithm* of Yang, Neiger, and Gafni [117]: it alternates an eventually-successful agreement protocol with an eventually-successful termination-detection protocol. The protocols and underlying models, however, are quite different.

3.3 Topological Model

In our model, a set of $n+1$ *processes* communicate by message-passing. An initial or final state of a process is modeled as a *vertex*, $\langle P_i, v\rangle$, a pair consisting of a process id P_i and a value v (either input or output). (Sometimes a vertex is labeled only with a process id.)

Definition 3.1. *A d-dimensional simplex $S^d = (s_0, \ldots, s_d)$, called $d-simplex$, is a set of $d+1$ vertexes that model mutually compatible initial or final process states. We say that $s_0, \ldots, s_d$* span *S^d. Simplex T is a (proper) $face$ of S^d if the vertexes of T form a (proper) subset of the vertexes of S^d.*

We use $\dim(S)$ for the dimension of S, and *ids*(S) for the set of process ids labeling vertexes of S.

If X is a subset of the process ids labeling a simplex S, then $S\backslash X$ is the face of S labeled with simplexes not in X.

Definition 3.2. *A* simplicial complex *(or complex) is a set of simplexes closed under containment and intersection. The* dimension *of a complex is the highest dimension of any of its simplexes. $\mathcal{L}$ is a* subcomplex *of $\mathcal{K}$ if every simplex of $\mathcal{L}$ is a simplex of $\mathcal{K}$.*

We sometimes indicate the dimension of a simplex or complex as a superscript.

A key idea is the concept of a *pseudosphere* [66], a simple combinatorial structure in which each process from a set of processes is independently assigned a value from a set of values. Pseudospheres have a number of nice combinatorial properties (for example, they are closed under intersection), but their principal interest lies in the observation that the behavior of the protocols we consider can be characterized as simple compositions of pseudospheres.

Definition 3.3. *Let P^n be the simplex in which each vertex is labeled with a process id. and $U_0, \ldots, U_n$ a sequence of finite sets. The* pseudosphere *$\psi(P^n; U_0, \ldots, U_n)$ is the following complex. Each vertex is a pair $\langle P_i, u_i\rangle$, where P_i is a process id (a vertex of P^n) and $u_i \in U_i$. Vertexes $\langle P_{i_0}, u_{i_0}\rangle, \ldots, \langle P_{i_\ell}, u_{i_\ell}\rangle$ span a simplex of $\psi(P^n; U_0, \ldots, U_n)$ if and only if the P_i are distinct. A pseudosphere in which all U_i are equal to U is simply written $\psi(P^n; U)$.*

Definition 3.4. *A* protocol *is a program in which each process starts with a private input value, communicates with the other processes via message-passing, and eventually halts with a private output value. Processes may crash, halting in the middle of the protocol, and messages in transit may be delayed for arbitrary finite durations. Processes may use failure detectors to decide when to stop waiting for messages. Without loss of generality, we restrict attention to* full-information *protocols in which each process sends its entire state in each message.*

Definition 3.5. *Any protocol has an associated* protocol complex $\mathcal{P}$, *defined as follows. Each vertex is labeled with a process id and a possible local state for that process. Vertexes* $\langle P_0, v_0 \rangle, \ldots, \langle P_d, v_d \rangle$ *span a simplex of* $\mathcal{P}$ *if and only if there is some protocol execution in which* $P_0, \ldots, P_d$ *finish the protocol with respective local states* $v_0, \ldots, v_d$. *Each simplex thus corresponds to an equivalence class of executions that "look the same" to the processes at its vertexes. The protocol complex* $\mathcal{P}$ *depends both on the protocol and on the timing and failure characteristics of the model.*

It is convenient to treat a protocol complex as an *operator* carrying input simplexes or complexes to protocol complexes.

Informally, a complex is k-connected if it has no holes in dimensions k or less. More precisely:

Definition 3.6. *A complex* $\mathcal{K}$ *is* k-connected *if every continuous map of the* ℓ*-sphere to* $\mathcal{K}$ *can be extended to a continuous map of the* $(\ell + 1)$*-disk [112, p. 51], for all* $0 \leq \ell \leq k$. *By convention, a complex is* (-1)-connected *if it is non-empty, and any complex is vacuously* ℓ*-connected for* $\ell < -1$.

This definition of k-connectivity may seem cumbersome, but fortunately we can do all our reasoning in a combinatorial way, using the following elementary consequence of the Mayer-Vietoris sequence [95, p. 142].

Theorem 3.7. *If* $\mathcal{K}$ *and* $\mathcal{L}$ *are complexes such that* $\mathcal{K}$ *and* $\mathcal{L}$ *are* k*-connected, and* $\mathcal{K} \cap \mathcal{L}$ *is* $(k-1)$*-connected, then* $\mathcal{K} \cup \mathcal{L}$ *is* k*-connected.*

As a base case for all such inductions, any simplex S^n is n-connected.

Theorem 3.8 ([66]). *If* $U_0, \ldots, U_m$ *are non-empty, then* $\psi(S^m; U_0, \ldots, U_m)$ *is* $(m-1)$*-connected.*

Finally, the notion of k-connectivity lies at the heart of all known lower bounds for k-set agreement. We now give a general theorem linking $(k-1)$-connectivity with impossibility of k-set agreement, originally stated in [65]. Note that this theorem is model-independent in the sense that it depends on the connectivity properties of protocol complexes, not on explicit timing or failure properties of the model.

Theorem 3.9. *Let* $\mathcal{I}$ *be the standard input complex* $\psi(P^n, V)$, *where* $|V| > k$. *If the complex* $\mathcal{P}(\mathcal{I})$ *is* $(k-1)$*-connected, then* $\mathcal{P}$ *cannot solve* k*-set agreement in the presence of* f *failures.*

3.4 Models of Computation

Without loss of generality, we assume that processes execute in *asynchronous rounds*: at round r, a process broadcasts a message containing its state to all of the others, and then waits until it receives round-r messages from all unsuspected processes (including itself). Messages are *full-information*, containing each process's complete state, including a history of all messages sent and received up to that point. Failure detectors satisfy *strong completeness*: if after some point Q never sends a message to P, then P will eventually suspect Q and stop waiting for that message.

The *basic* model of computation guarantees only that each non-faulty process at round r will eventually receive round-r messages from at least $n - f + 1$ processes.

Definition 3.10. *A message-passing model satisfies* causality *if it satisfies the following condition. If*

1. *process P sends a message p to all processes,*
2. *Q receives p and later sends q to all processes, and*
3. *R receives q at round r,*

then R receives p at a round less than or equal to r.

We claim that causality adds nothing to the computational power of the basic model.

Lemma 3.11. *If a protocol solves k-set agreement for a given x, q, and f in the basic model with causality, then the same is true for the basic model (without causality).*

Proof. Let P, Q, and R be processes as described in Definition 3.10. Suppose R receives message q at round r without receiving p at a round less than or equal to r. Because messages are full-information, R can extract p from r, and act as if it had also received p. $\square$

It follows that if there is no k-consensus protocol for given x, q, and f in the basic model, then there is no such protocol in the basic model with causality.

A message-passing model satisfies *eventual delivery* if every message is eventually delivered to every non-faulty process.

Definition 3.12. *The* standard model *is the basic model with causality and eventual delivery.*

We claim that eventual delivery adds nothing to the computational power of the basic model with causality.

Lemma 3.13. *If a protocol solves k-set agreement for a given x, q, and f in the standard model, then the same is true for the basic model with causality.*

Proof. Suppose we have a protocol $\mathcal{P}$ that always terminates in the standard model, but has an infinite execution in the basic model with causality (caused by undelivered messages). Define a *lost* message to be one that is never delivered to some non-faulty process. Let L be the set of processes that send a lost message, M the set of those that don't, and F the set of those that fail. (L and M are disjoint, but may overlap F.)

Once a process P in L sends a lost message, it can never send another message to any process in M, because the later message would be forwarded to every non-faulty process, and causality would force delivery of the earlier "lost" message to every non-faulty process. Once P falls silent to M, strong completeness implies it will eventually be suspected by every process in M.

We claim that $|L \cup F| \leq f$. Wait until every process in L has sent a lost message and fallen silent to M, and every process in F has failed and fallen silent to M. The model ensures that messages from at least $n - f + 1$ processes continue to be delivered to each process in M, so the missing processes can only come from $L \cup F$.

The infinite execution of $\mathcal{P}$ in the basic model with causality is thus indistinguishable, to the processes in M, from an infinite execution in the standard model where processes in L fail instead of sending the first lost message. Since the protocol terminates in the second model, it terminates in the first. □

It is convenient to prove our lower bounds in the basic model.

Corollary 3.14. *If there is no protocol that solves k-set agreement for a given x, q, and f in the basic model, then the same is true for the standard model.*

3.5 Perpetual Weak (x, q)-Accuracy

Without loss of generality, assume that each process's input value is an integer from a set V of size greater than k. Let $X_0, \ldots, X_q$ be the clusters, where each X_i has size x_i, and clusters are indexed by decreasing size ($x_i \geq x_{i+1}$). Let x be the combined size of the k largest clusters (or the q largest, if $q < k$). More precisely, let $\kappa = \min(k, q)$.

$$x = \sum_{i=0}^{\kappa-1} x_i$$

Let $\mathcal{I}^n$ be the input complex in which each process is independently given an input value from V. This complex is a pseudosphere: $\mathcal{I}^n = \psi(P^n; V)$.

Lower bounds apply even if we restrict the behavior of the adversary, so we will restrict our attention to executions in which processes run in (asynchronous) rounds: each process repeatedly broadcasts its message and waits for $n - f + 1$ messages to arrive.

3.5.1 No Clusters

Let $\mathcal{D}$ be the operator that corresponds to a one-round execution in the basic model in which all failure detectors satisfy only strong completeness. Operationally, each process receives messages containing full state information from at least $n - f - 1$ processes. (For simplicity, we assume a process may or may not receive a message from itself.) Combinatorially, after one round each process's vertex is labeled with a face of I^n of dimension at least $n - f$. The result of the the one-round operator is thus a pseudosphere:

$$\mathcal{D}(I^n) = \psi(I^n; U_{n-f}),$$

where $U_{n-f} = \{T | T \subset I^n \text{ and } \dim(T) \geq n - f\}$, the set of faces of I^n of dimension at least $n - f$. It follows that $\mathcal{D}(I^n)$ is $(f - 1)$-connected.

Let $\mathcal{D}^r(I^n)$ denote the r-round protocol complex on input simplex I^n. Recall that P^n is an n-simplex where vertex i is labeled with process id i, $\psi(P^n; \{u_0\}, \ldots, \{u_n\})$ is a simplex where vertex i is labeled with process id i and value u_i.

Theorem 3.15. *Let $I^n = \psi(P^n; \{u_0\}, \ldots, \{u_n\})$. If $\mathcal{D}^r(I^n)$ is $(f - 1)$-connected, so is $\mathcal{D}^r(\psi(P^n; U_0, \ldots, U_n))$ for non-empty sets $U_0, \ldots, U_n$.*

Proof. By induction on f. For base case, let $f = 0$. $\mathcal{D}^r(\psi(P^n; U_0, \ldots, U_n))$ is non-empty, and is therefore $(f-1)$-connected.

Define the following partial order on sequences of sets: $(U_0, \ldots, U_m) \prec (V_0, \ldots, V_m)$ if each $U_i \subseteq V_i$, and for at least one set, the inclusion is strict.

We argue by induction on the partially-ordered sequences. For the base case, $\mathcal{D}^r(I^n)$ is $(f-1)$-connected by hypothesis.

For the induction step for the set sequences, let $U_0, \ldots, U_m$ be a non-minimal sequence of sets, and assume the claim for every preceding sequence. Since the U_i are not all singleton sets, there must be some index i such that $U_i = V_i \cup \{v\}$, where V_i is non-empty. It follows that

$$\mathcal{D}^r(\psi(P^m; U_0, \ldots, U_m)) = \mathcal{K} \cup \mathcal{L},$$

where

$$\mathcal{K} = \mathcal{D}^r(\psi(P^m; U_0, \ldots, V_i, \ldots, U_m)),$$

and

$$\mathcal{L} = \mathcal{D}^r(\psi(P^m; U_0, \ldots, \{v\}, \ldots, U_m)).$$

By the induction hypothesis for the sets, both $\mathcal{K}$ and $\mathcal{L}$ are $(f-1)$-connected.

A vertex is in $\mathcal{K}$ and $\mathcal{L}$ if and only if the process id P_i does not appear in any vertex label. The complex $\mathcal{K} \cap \mathcal{L}$ is the result of an r-round, f-failure execution in which P_i fails before sending any messages. Equivalently, $\mathcal{K} \cap \mathcal{L} = \mathcal{D}^r(I^n \setminus \{P_i\})$, the result of an r-round, $(f-1)$-failure execution over all processes except P_i. By the induction hypothesis on f, $\mathcal{K} \cap \mathcal{L}$ is $(f-2)$-connected, and by Theorem 3.7, $\mathcal{K} \cup \mathcal{L}$ is $(f-1)$-connected. □

Theorem 3.16. *$\mathcal{D}^r(I^n)$ is $(f-1)$-connected.*

Proof. We argue by induction on r. For the base case, $r = 1$, and $\mathcal{D}(I^n)$ is a pseudosphere, and therefore $(n-1)$-connected and also $(f-1)$-connected.

For the induction step, assume $\mathcal{D}^{r-1}(I^n)$ is $(f-1)$-connected. By Theorem 3.15, $\mathcal{D}^{r-1}$ applied to any pseudosphere over I^n is also $(f-1)$-connected. In particular, $\mathcal{D}(I^n)$ is a pseudosphere, so $\mathcal{D}^{r-1}(\mathcal{D}(I^n)) = \mathcal{D}^r(I^n)$ is $(f-1)$-connected. □

3.5.2 Single Cluster

We now consider the case where there is a single cluster X of x processes, and a single process P_0 never suspected by any process in X. Let $\mathcal{D}_x$ be the corresponding operator. Operationally, each process receives messages containing full state information from at least $n-f-1$ processes, and every process in X receives a message from P_0. The result of the one-round operator is thus the pseudosphere:

$$\mathcal{D}_x(I^n) = \psi(I^n; U_0, \ldots, U_n),$$

where

$$U_i = \begin{cases} \{F | F \subset I^n, \dim(F) \geq n-f, \text{ and } P_i \in \mathit{ids}(F)\} & \text{if } P_i \in X \\ \{F | F \subset I^n \text{ and } \dim(F) \geq n-f\} & \text{otherwise} \end{cases}$$

It follows that $\mathcal{D}_x(I^n)$ is $(f-1)$-connected.

Let $\mathcal{D}_x^r(I^n)$ denote the r-round protocol complex on input simplex I^n for a given cluster X.

Theorem 3.17. *Let $I^n = \psi(P^n; \{u_0\}, \ldots, \{u_n\})$. If $f \geq x-1$, and $\mathcal{D}_x^r(I^n)$ is $(f-x)$-connected, so is $\mathcal{D}_x^r(\psi(P^n; U_0, \ldots, U_n))$ for non-empty sets $U_0, \ldots, U_n$.*

Proof. We proceed by induction on f. For the base case, let $f = x - 1$. $\mathcal{D}_x^r(I^n)$ is non-empty, hence $(f - x)$-connected.

Assume the claim for fewer than f failures. As before, let $\prec$ be the partial order on sequences of sets. We argue by induction on the partially-ordered sequences. For the base case, $\mathcal{D}_x^r(I^n)$ is $(f - x)$-connected by hypothesis.

For the induction step for the set sequences, let $U_0, \ldots, U_m$ be a non-minimal sequence of sets, and assume the claim for every preceding sequence. Since the U_i are not all singleton sets, there must be some index i such that $U_i = V_i \cup \{v\}$, where V_i is non-empty. It follows that

$$\mathcal{D}_x^r(\psi(P^m; U_0, \ldots, U_m)) = \mathcal{K} \cup \mathcal{L}$$

where

$$\mathcal{K} = \mathcal{D}_x^r(\psi(S^m; U_0, \ldots, V_i, \ldots, U_m)),$$

and

$$\mathcal{L} = \mathcal{D}_x^r(\psi(S^m; U_0, \ldots, \{v\}, \ldots, U_m)).$$

By the induction hypothesis for the sets, both $\mathcal{K}$ and $\mathcal{L}$ are $(f - x)$-connected.

A vertex is in $\mathcal{K}$ and $\mathcal{L}$ if and only if P_i does not appear in any vertex label. There are two cases to consider. Suppose $P_i = P_0$, the process trusted by every process in X. Every process in X receives a message from P_0, so no process in X appears in the label of any vertex in $\mathcal{K} \cap \mathcal{L}$. If $f \geq x$, the complex $\mathcal{K} \cap \mathcal{L}$ is the result of an r-round, f-failure execution of $\mathcal{D}_x^r$ in which each process in X fails before sending any messages. Equivalently, because $f - x \geq 0$,

$$\mathcal{K} \cap \mathcal{L} = \mathcal{D}^r(I^n \backslash X),$$

the result of an r-round, $(f - x)$-failure execution of $\mathcal{D}^r$ over the face of the input simplex labeled with the $n - x + 1$ processes not in X. By Theorem 3.16, $\mathcal{K} \cap \mathcal{L}$ is $(f - x - 1)$-connected. If $f = x - 1$, then $\mathcal{K} \cap \mathcal{L}$ is empty. Notice that $f - x - 1 = -2$, so $\mathcal{K} \cap \mathcal{L}$ is (trivially) $(f - x - 1)$-connected.

If P_i is distinct from P_0, then P_i does not appear in the label of any vertex in $\mathcal{K} \cap \mathcal{L}$. The complex $\mathcal{K} \cap \mathcal{L}$ is thus the result of an r-round, f-failure execution of $\mathcal{D}_x^r$ in which P_i fails before sending any messages. If P_i is in X, then this complex is equivalent to

$$\mathcal{K} \cap \mathcal{L} = \mathcal{D}_{x-1}^r(I^n \backslash \{P_i\}),$$

the result of an r-round, $(f - 1)$-failure execution of $\mathcal{D}_{x-1}^r$ over the face of the input simplex containing every process except P_i. Let $x' = x - 1$ and $f' = f - 1$. By the induction hypothesis for f, $\mathcal{K} \cap \mathcal{L}$ is $(f' - x')$-connected, and therefore $(f - x)$-connected, and $(f - x - 1)$-connected.

If P_i is not in X, then this complex is equivalent to

$$\mathcal{K} \cap \mathcal{L} = \mathcal{D}_x^r(I^n \backslash \{P_i\}).$$

Let $x' = x$ and $f' = f - 1$. By the induction hypothesis for f, $\mathcal{K} \cap \mathcal{L}$ is $(f' - x')$-connected, and therefore $(f - x - 1)$-connected.

Since $\mathcal{K}$ and $\mathcal{L}$ are each $(f - x)$-connected, and $\mathcal{K} \cap \mathcal{L}$ is $(f - x - 1)$-connected, it follows from Theorem 3.7 that $\mathcal{K} \cup \mathcal{L}$ is $(f - x)$-connected. □

Theorem 3.18. *If $f \geq x - 1$, $\mathcal{D}_x^r(I^n)$ is $(f - x)$-connected.*

Proof. We argue by induction on r. For the base case, $r = 1$, and $\mathcal{D}_x(I^n)$ is a pseudosphere, and therefore $(n-1)$-connected and also $(f-x)$-connected.

For the induction step, assume $\mathcal{D}_x^{r-1}(I^n)$ is $(f-x)$-connected. By Theorem 3.17, $\mathcal{D}_x^{r-1}$ applied to any pseudosphere over I^n is also $(f-x)$-connected. In particular, $\mathcal{D}_x(I^n)$ is a pseudosphere, so $\mathcal{D}_x^{r-1}(\mathcal{D}_x(I^n)) = \mathcal{D}_x^r(I^n)$ is $(f-x)$-connected. □

3.5.3 Multiple Clusters

We now consider the case where there are multiple clusters $X_0, \ldots, X_{q-1}$, where each X_j contains a correct process never suspected by any process in X_j. Each X_j has size x_j. Index the processes so that P_j is not suspected by any process in X_j, for $0 \le j < q$.

Let $\mathcal{D}_{x,q}$ be the corresponding one-round operator. Operationally, each process receives messages containing full state information from at least $n-f-1$ processes, subject to the condition that every process in X_j receives a message from P_j, for $0 \le j < q$. Combinatorially, after one round each process's vertex is labeled with a face of I^n of dimension at least $n-f$, where the faces labeled processes in X_j include P_j's vertex. The result of the the one-round operator is thus a pseudosphere:

$$\mathcal{D}_{x,q}(I^n) = \psi(I^n; U_0, \ldots, U_n),$$

$$U_i = \begin{cases} \{F | F \subset I^n \text{ and } \dim(F) \ge n-f \text{ and } P_j \in \mathit{ids}(F)\} & \text{if } P_i \in X_j \\ \{F | F \subset I^n \text{ and } \dim(F) \ge n-f\} & \text{otherwise} \end{cases}$$

It follows that $\mathcal{D}_{x,q}(I^n)$ is $(f-1)$-connected.

Recall that $\kappa = \min(q,k)$.

Theorem 3.19. *Let $I^n = \psi(P^n; \{u_0\}, \ldots, \{u_n\})$. If $\mathcal{D}_{x,q}^r(I^n)$ is $(f-x+\kappa-1)$-connected, so is $\mathcal{D}_{x,q}^r(\psi(P^n; U_0, \ldots, U_n))$ for non-empty sets $U_0, \ldots, U_n$.*

Proof. We argue by induction on f. For the base case, let $f = x-\kappa$. $\mathcal{D}_{x,q}^r(\psi(S^n; U_0, \ldots, U_n))$ is non-empty, hence $(f-x+\kappa-1)$-connected.

Assume the claim for fewer than f failures. As before, let $\prec$ be the partial order on sequences of sets. We argue by induction on the partially-ordered sequences. For the base case, $\mathcal{D}_{x,q}^r(I^n)$ is $(f-x+\kappa-1)$-connected by hypothesis.

For the induction step for the set sequences, let $U_0, \ldots, U_m$ be a non-minimal sequence of sets, and assume the claim for every sequence that precedes $U_0, \ldots, U_m$ in the partial order. Since the U_i are not all singleton sets, there must be some index i such that $U_i = V_i \cup \{v\}$, where V_i is non-empty. It follows that

$$\mathcal{D}_{x,q}^r(I^n) = \mathcal{K} \cup \mathcal{L},$$

where

$$\mathcal{K} = \mathcal{D}_{x,q}^r(\psi(P^m; U_0, \ldots, V_i, \ldots, U_m)),$$

and

$$\mathcal{L} = \mathcal{D}_{x,q}^r(\psi(P^m; U_0, \ldots, \{v\}, \ldots, U_m)).$$

By the induction hypothesis for the sets, both $\mathcal{K}$ and $\mathcal{L}$ are $(f-x+\kappa-1)$-connected.

A vertex is in $\mathcal{K}$ and $\mathcal{L}$ if and only if P_i does not appear in any vertex label. Suppose $i < q$, meaning P_i is trusted by every process in X_i. Every process in X_i receives a message from P_i, so no vertex with a label in X_i appears in $\mathcal{K} \cap \mathcal{L}$. The complex $\mathcal{K} \cap \mathcal{L}$ is thus the result of an r-round, f-failure execution of $\mathcal{D}_{X,q}^r$ in which each process in X_i fails before sending any messages.

If $i < \kappa$, then this complex is equivalent to

$$\mathcal{K} \cap \mathcal{L} = \mathcal{D}^r_{x-x_i,q-1}(I^n \backslash X_i),$$

the result of an r-round, $(f-x_i)$-failure execution over the $n-x_i$ processes not in X_i. Let $f' = f - x_i$, $x' = x - x_i$, $q' = q-1$, $k' = k-1$, and $\kappa' = \min(q', k') = \kappa - 1$. Note that x' is the combined size of the $q'-1$ largest clusters, so by the induction hypothesis for f, $\mathcal{K} \cap \mathcal{L}$ is $(f' - x' + \kappa' - 1)$-connected, and is therefore $(f - x + \kappa - 2)$-connected.

Suppose $i \geq \kappa$. Let $f' = f - x_i$, $q' = q-1$, $k' = k-1$, and $\kappa' = \min(q', k') = \kappa - 1$. Let $x' = x - x_{\kappa-1}$, the combined size of the $\kappa - 1$ largest clusters. Notice that $x' \leq x - x_i$. By the induction hypothesis for f, $\mathcal{K} \cap \mathcal{L}$ is $(f' - x' + \kappa' - 1)$-connected. Because

$$f' - x' + \kappa' - 1 = f - x_i - (x - x_{\kappa-1}) + \kappa - 1 - 1 \geq f - x - \kappa - 2,$$

$\mathcal{K} \cap \mathcal{L}$ is $(f - x + \kappa - 2)$-connected.

Finally, suppose P_i is not the trusted process for any X_i. No vertex with a label containing P_i appears in $\mathcal{K} \cap \mathcal{L}$. The complex $\mathcal{K} \cap \mathcal{L}$ is thus the result of an r-round, f-failure execution of $\mathcal{D}^r_{X,q}$ in which P_i fails before sending any messages. If P_i is in $X_0 \cup \cdots \cup X)\kappa - 1$, then

$$\mathcal{K} \cap \mathcal{L} = \mathcal{D}^r_{x-1,q}(I^n \backslash \{P_i\}),$$

the result of an r-round, $(f-1)$-failure execution over the processes distinct from P_i. Let $f' = f-1$, and $x' = x-1$, the combined size of the κ largest clusters. By the induction hypothesis for f, $\mathcal{K} \cap \mathcal{L}$ is $(f' - x' + \kappa - 1)$-connected, and therefore $\mathcal{K} \cap \mathcal{L}$ is $(f - x + \kappa - 1)$-connected. If P_i is not in $X_0 \cup \cdots \cup X)\kappa - 1$, then

$$\mathcal{K} \cap \mathcal{L} = \mathcal{D}^r_{x,q}(I^n \backslash \{P_i\}),$$

the result of an r-round, $(f-1)$-failure execution over the processes distinct from P_i. By the induction hypothesis for f, this complex is $(f - x + \kappa - 2)$-connected,

In all cases, $\mathcal{K} \cap \mathcal{L}$ is at least $(f - x + q - 2)$-connected, and so by Theorem 3.7, $\mathcal{K} \cup \mathcal{L}$ is $(f - x + \kappa - 1)$-connected. $\square$

Theorem 3.20. *$\mathcal{D}^r_{x,q}(I^n)$ is $(f - x + \kappa - 1)$-connected.*

Proof. We argue by induction on r. For the base case, $r = 1$, and $\mathcal{D}_{x,q}(I^n)$ is a pseudosphere, and therefore $(n-1)$-connected and also $(f - x + \kappa - 1)$-connected.

For the induction step, assume $\mathcal{D}^{r-1}_{x,q}(I^n)$ is $(f-x+\kappa-1)$-connected. By Theorem 3.17, $\mathcal{D}^{r-1}_{x,q}$ applied to any pseudosphere over I^n is also $(f - x + \kappa - 1)$-connected. In particular, $\mathcal{D}_{x,q}(I^n)$ is a pseudosphere, so $\mathcal{D}^{r-1}_x(\mathcal{D}_{x,q}(I^n)) = \mathcal{D}^r_{x,q}(I^n)$ is $(f-x+\kappa-1)$-connected. $\square$

Theorem 3.21. *$\mathcal{D}^r_{x,q}(\mathcal{I}^n)$ is $(f - x + \kappa - 1)$-connected.*

Proof. By Theorem 3.18, $\mathcal{D}^r_{x,q}$ applied to any pseudosphere over S^n is also $(f-x+\kappa-1)$-connected. In particular, $\mathcal{I}^n$ is a pseudosphere over P^n, so $\mathcal{D}^r_{x,q}(\mathcal{I}^n) = \mathcal{D}^r_{x,q}(\psi(S^n; V))$ is $(f - x + \kappa - 1)$-connected. $\square$

Theorem 3.22. *There exist protocols solving k-set agreement for $n+1$ processes with failure detectors of type $S_{x,q}$ if and only if*

$$f < \begin{cases} k + x - q & q \leq k \\ x & \textit{otherwise} \end{cases}$$

Proof. By Theorems 3.9 and 3.21, a protocol exists only if $f < k + x - \kappa$. We start by generalizing the Terminating Weak Agreement (TWA) protocol of Mostéfaoui and Raynal [94] to encompass q sets X_i, instead of just one.

The TWA_q protocol for process p appears in Figure 3.1. We are given a set Y of processes, a unique id u, and an estimate e. The protocol iterates over all subsets Q of Y of size q. At each round, the protocol generates a round number v. The concatenation $\langle u, v\rangle$ is used as a unique label for messages. If p is in Q, then it broadcasts its estimate to the other processes in X Otherwise, p waits for a message from a process in Q, labeled with id $\langle u, v\rangle$. If it receives such a message, it adopts the estimate in from that message. If the process eventually suspects all the processes in Q, then it starts the next round with the same estimate.

We claim that if TWA_q is called with Y equal to the set of clusters, then the number of estimates will be no larger than q. At some point during the protocol, the set Q will contain exactly the processes P_i, where P_i is the correct process perpetually trusted by every process in X_i. During that round, no process will suspect every process in Q, because every process is in some X_i, and will trust $P_i \in Q$. Each process will adopt the estimate of some process in Q, so the total number of distinct estimates among the processes in X is at most q.

We now generalize the S_x protocol of Mostéfaoui and Raynal. The $S_{x,q}$ protocol is the same as S_x, except we take $m = k + x - q$ instead of $m = k + x - 1$, where m is the number of processes given to the TWA_q protocol. From Mostéfaoui and Raynal [94], it is straightforward to see that this slightly modified protocol solves k-set agreement in asynchronous distributed systems equipped with failure detectors from the class $S_{x,q}$. □

Theorem 3.23. *There exist protocols solving k-set agreement for $n+1$ processes with failure detectors of type $\diamond S_{x,q}$ if and only if $f < \min(\frac{n+1}{2}, k + x - q)$.*

Proof. Both $S_{x,q}$ and $\diamond S$ are at least as strong as $\diamond S_{x,q}$, and Chandra and Toueg [28] show that $f < \frac{n+1}{2}$ for $\diamond S$. Moreover, by Theorem 3.22, $f < k + x - q$ for $S_{x,q}$. It follows that $f < min(\frac{n+1}{2}, k + x - q)$ for $\diamond S_{x,q}$. The matching protocol is given in the next section. □

3.6 Eventual Weak (x, q)-Accuracy

```
int TWA(int p, Set Y, int e, int u) {
  int v = 0;
  for each subset Q of size q in Y {
    if (p is in Q) { // I am a coordinator
      broadcast new TWAMessage(r, e);
    } else {          // I am not coordinator
      try {
        // receive round (u,v) message from anyone in Q
        Message m = receive(Q, <u,v>);
        e = m.estimate;            // take other's estimate
      } catch (suspectedException e) {
          skip if all process in Q suspected
      }
    v = v + 1;
    }
  }
  return e;
}
```

Figure 3.1: The TWA_q Protocol

In this section, we present a novel protocol that matches our lower bound for failure detectors in the class $\diamond S_{x,q}$. We start with a slightly modified version of the *Terminating Weak Agreement* (TWA) protocol of Mostéfaoui and Raynal, illustrated by pseudocode in Figure 3.1. This protocol takes a set of m participating processes, an initial value for each participating process, a round number, and the ID of the calling process. It considers all possible orderings of the set of m participating processes (which is relevant just when $q > 1$). It guarantees that if the set of participating processes includes q sets X_i of x_i processes such that some correct process in X_i is not suspected by any process in X_i, then at most $m - x + q$ values are decided.

It is straightforward to extend the TWA protocol to solve k-set agreement for the $S_{x,q}$ failure detector: simply run TWA for each subset of $m = \min(n + 1, k + x - q)$ processes (Figure 3.2). Each process has an *estimate*, originally its input value. Each iteration introduces no new estimates. Each process chooses a new estimate at the end of each round, and retains the estimate it decided in the previous iteration. At some point, the m processes will encompass the processes in X, and the m processes will henceforth agree on at most $k = m - x + q$ distinct estimates for at least one of the orderings (namely, the one where the trusted processes are positioned at the end). Every process not participating in that round's TWA protocol waits for a message from a participant (which will arrive reducing the maximum number of distinct estimates from $n + 1$ to $k = m - x + q$). See Mostéfaoui and Raynal [94] for a more complete discussion.

For our new $\diamond S_{x,q}$ protocol, we repeatedly run the TWA-based protocol. Eventually, when all failure detectors have achieved weak (x, q)-accuracy, each subsequent iteration of the TWA-based algorithm will yield k or fewer values. The challenge is to detect when the TWA-based algorithm has converged.

We cycle through all permutations of the $n + 1$ processes. A *low-order* process in a

```
int SxqAgree(int id, int estimate) {
  int m = min(n+1, k + x - q);
  // try all sets of size m
  for (round = 0; round < C(n+1, m); round++) {
    // next subset of m processes
    Set particip = ProcessSet.subset(round);
    if (id element of particip) {  // I'm in the group
      estimate = TWAq(id, particip, estimate, round);
      broadcast new Message(round, estimate);
    } else {
      Message message = receive(round);
      // take other's value
      estimate = message.estimate;
    }
  }
  return estimate;
}
```

Figure 3.2: k-set agreement protocol for $S_{x,q}$

permutation is one with rank less than or equal to $\lfloor \frac{n+1}{2} \rfloor + 1$, and the rest are *high-order* processes. Each process broadcasts its estimate, waits to receive $\lfloor \frac{n+1}{2} \rfloor + 1$ messages, and changes its estimate to the estimate from the least-ranked process in the current permutation. Because $f < \frac{n+1}{2}$, each high-order process will receive a message from a low-order process, so at the end of the round, every process will have switched to an estimate from a low-order process. If we can determine that the low-order processes had at most k distinct estimates at the start of the round, then all processes will have at most k estimates at the end of the round.

Each process includes in its message a history of its estimates at the start of all earlier rounds. Suppose, in round r, a process P receives messages from a set S of $\lfloor \frac{n+1}{2} \rfloor + 1$ processes. Let $s \leq r$ be the most recent round, if any, for which S was the set of low-order processes for the permutation at round s. P checks the histories received to determine whether the processes in S had at most k distinct values at round s. If so, the protocol has converged, and P can halt. The protocol is illustrated in Figure 3.3.

It is worth emphasizing that the `DiamondAgree` protocol does not actually depend on the TWA-based protocol, or even on $S_{x,q}$. It requires only (1) that the embedded protocol does not increase the set of original estimates, (2) it eventually solves k-set agreement, and (3) that there are fewer than $\lfloor \frac{n+1}{2} \rfloor$ failures.

Note that the complexity of rounds after accuracy is achieved may be lowered from exponential to polynomial if one changes the size of the low-order set (and the set of received messages) from $\lfloor \frac{n+1}{2} \rfloor + 1$ to $(n+1) - \min(\lfloor \frac{n+1}{2} \rfloor, k + x - q) + 1$ and considers for the number of rounds all possible combinations for the low-order sets instead of all (n+1)! permutations of processes. However, this only works when $k + x - q$ is constant.

Note also that getting rid of the anonymity of processes lowers the complexity for $S_{x,q}$ from exponential to polynomial, since the combination and ordering of processes are no longer needed in the TWA protocol. However, it seems that this does not help in reducing

```
public int DiamondAgree(int id, int estim) {
  for (int r = 0; ; r++) {  // run until accuracy achieved
    estim = SxqAgree(id, estim);
    // cycle through permutations
    for (int p = 0; p < (n+1)!; p++) {
      Perm perm = new Perm(p); // construct permutation
      broadcast new Message(r, p, id, estim, hist);

      // wait for ((n+1)/2)+1 messages
      MessageSet mSet =receive(r, p, ((n+1)/2)+1);

      // take estimate from low-order process
      estim = mSet.getLowOrderEstimate();

      // when were these processes all low-order?
      ProcessSet pSet = mSet.getProcesses();
      int lowOrderRound = Perm.firstLowOrder(pSet);
      if (lowOrderRound <= round) { // has it happened yet?
        // get low-order estimates from that round
        EstimateSet eSet = mSet.getEstimates(lowOrderRound);
        if (eSet.size() <= k)
          broadcast new SuccessMessage();
          return estim;
      }
    }
  }
}
```

Figure 3.3: k-set agreement protocol for $\diamond S_{x,q}$

the complexity for $\diamond S_{x,q}$.

Theorem 3.24. *Let $f < min(\frac{n+1}{2}, k+x-q)$. The protocol illustrated in Figure 3.3 solves k-set agreement in asynchronous distributed systems equipped with failure detectors from class $\diamond S_{x,q}$.*

Proof. The proof has three parts. Note that $m = min(n+1, k+x-q)$, as in the code.

Validity follows from validity of the $S_{x,q}$ protocol, and because every estimate is always set to another process's estimate.

Termination. Because $f < m = min(n+1, k+x-q)$, each TWA instance terminates and at least one process from the participating set broadcasts an estimate. Moreover, no process waits forever for $\lfloor \frac{n+1}{2} \rfloor + 1$ messages.

The eventual (x, q)-accuracy property ensures that at some point there are q sets X_i of x_i processes such that some correct process in X_i is not suspected by any process in X_i. Consider the first round for which this property holds. The first subsequent execution of the $S_{x,q}$ agreement protocol will reduce the number of distinct estimates to no more than k. After the $S_{x,q}$ protocol execution, the processes run through the permutations. At some round, some non-faulty process must receive messages from the processes that were the low-order processes for some permutation that occurred after the $S_{x,q}$ protocol execution, but before the current permutation. Checking the histories, that process will detect that the low-order processes had k or fewer distinct estimates, and the protocol will terminate when that process broadcasts an announcement.

Agreement. The protocol terminates if and only if there is a correct process that identifies an earlier permutation such that there were at most k distinct estimates among the low-order processes. Because every process sets its estimate to an estimate from a low-order process, there can be at most k distinct estimates among all processes. $\square$

Chapter 4

Robust Crash Transformations

Here we investigate computational models with stabilizing properties. Such models include e.g. the partially synchronous model [48], where after some unknown global stabilization time the system complies to bounds on computing speeds and message delays, or the asynchronous model augmented with unreliable failure detectors [28], where after some unknown global stabilization time failure detectors stop making mistakes [18, 19].

Using protocol transformations we show that many (families of such) models are equivalent regarding solvability. We also analyze the efficiency of such transformations regarding not only the number of steps in a model M_1 necessary to emulate a step in a model M_2, but also the stabilization shift, which bounds the number of steps in M_2 required to provide properties of M_2 after the stabilization of M_1.

In the following, we begin in Section 4.1 by motivating and presenting related work. Then in Section 4.2 we make a precise description of the system models. In Section 4.3 we show the models equivalence with the help of automatic protocol transformations and in Section 4.4 we measure the efficiency of such transformations. Finally, in Section 4.5 we discuss work presented.

4.1 Motivation

We consider distributed message passing systems that are subject to crash failures. Due to the well-known impossibility result for deterministic consensus in asynchronous systems [52], a lot of research was done about adding assumptions to the asynchronous model in order to allow solving the problem. These include assumptions on the timing behavior of processes and communication links [42, 48] as well as assumptions on the capability of processes to retrieve information on failures of others [28].

Failure detectors encapsulate timing assumptions in a modular way. The previous sentence is stated in many research papers and sometimes even the required amount of synchrony to solve a problem is expressed via the weakest failure detector necessary (e.g. [59]). Interestingly, Charron-Bost et al. [29] have shown that in general, failure detectors do *not* encapsulate timing assumptions properly. For perpetual kind failure detectors as the 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 $\mathcal{P}$. 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. This is the first issue we address.

Another line of research considers "asymmetric" models in which timing assumptions need not hold at all links and all correct processes — as in [42, 48, 27] — but only for a subset of components in a system. This stems from the following question in [78]: Is there a model that allows implementing the eventually strong failure detector $\diamond\mathcal{S}$ (which can be reduced to the eventual leader oracle Ω), but does not allow to implement $\diamond\mathcal{P}$ (i.e., the eventually perfect failure detector, whose output stabilizes to complete information on remote process crashes)? Indeed, it was shown in [3] that such models exist. Since [3], much interest [2, 4, 85, 71] arose in weakening the synchrony assumption of models (or adding as little as possible to the asynchronous model) in order to be able to implement Ω. Regarding solvability, if such models are stronger than the asynchronous one, then these models would allow to solve all problems that can be solved with Ω but would not allow to solve problems where Ω is too weak. The second issue addressed here is thus whether the different spatial distributions of timing assumptions proposed make a difference in the set of problems which they allow to solve.

To tackle these challenges, we consider two main types of models: *abstract computational models* and *system models*. On one hand, *abstract computational models*, such as round-based models and failure detector based models, do not consider the timing behavior of distributed systems. For instance, round-based models restrict the sets of messages which have to be received in the round they were sent, while failure detector based models introduce axiomatic properties to guarantee access to information about failures. On the other hand, *system models*, such as the partially synchronous and eventually synchronous models, have explicit assumption on processing speeds and message delays. However, note that a property which is shared by all models we consider is that they are stabilizing, i.e., they restrict the communication in a distributed computation only from some unknown stabilization time on.

Finally, we introduce the notion of *algorithm transformations*, which we use to compare different models from both a solvability and an efficiency viewpoint.

4.1.1 Contribution

Expressiveness of Models. Here we show that the result of [29] is in fact limited to perpetual type failure detectors. To this end we introduce a new parametrized failure detector family, of which both $\diamond\mathcal{P}$ and $\diamond\mathcal{S}$ are special cases. Additionally, we define new parametrized (with respect to the number and distribution of eventually timely links and processes) partially and eventually synchronous model families. In terms of solvability, we show equivalence when instantiating both families with the same parameter k. As a corollary we show that the asynchronous system with $\diamond\mathcal{P}$ allows to solve the same problems as the classic partially synchronous model in [48], as well as that the asynchronous model augmented with Ω is equivalent to several source models, i.e., models where just the links from at least one process (the source) are timely.

Method. We introduce the notion of *algorithm transformations* for partially synchronous systems and for asynchronous systems augmented with failure detectors. While *transformations* (or their close relatives simulations [10]) are well understood in the context of synchronous as well as asynchronous systems, they have to the best of our knowledge never been studied before for partially synchronous systems.

In the case of synchronous systems, the simplifying assumption is made that no additional local computation and no number of messages that has to be sent for a simulation may lead to a violation of the lock-step round structure. It follows that layering of algorithms as proposed in [10] can be done very easily.

In contrast, for asynchronous systems no time bounds can be violated anyhow. Consequently, the coupling of the algorithm with 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 [82].

For partially synchronous systems, transformations are not straight forward. Based on primitives of lower models, primitives of higher models must be implemented in a more strongly coupled way than in asynchronous systems, while it has to be *ensured*[1] that the required timing properties are achieved. For that purpose, we define algorithm transformations, which we discuss in Section 4.2.6.

Cost. We also discuss the cost of these algorithm transformations, by examining two diverse measures. The first considers the required number of steps in one model in order to implement one step in the other one. To this end we introduce the notion of *B-bounded* transformations which means that any *step* of the higher model can be implemented by at most *B steps* of the lower model.

The second parameter considers how many steps are required to stabilize the implemented steps after the system has stabilized. For this we use the notion of *D-bounded shift*, which means that after the system stabilizes, the implemented steps stabilize at most after D steps. Further we introduce the notion of efficiency preserving: A transformation is *efficiency preserving*, if it is B-bounded, has D-bounded shift, and B and D are known in advance. We show between which pairs of models efficiency-preserving transformations exist.

[1]In sharp contrast to the synchronous case, where timing *is assumed to hold*.

4.2 System Models

Here we consider multiple models of distributed computations, which vary in their abstraction. For example, failure detector based models are at a higher level of abstraction than partially synchronous systems as they abstract away the timing behavior of systems [28], 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 — which can be done based on either timing assumptions [72] or on failure detector properties [28]. Whether these abstractions have the "cost" of losing relevant properties of the "lower level" model is the central question here.

We first turn to the common definitions, and then describe the specifics of each model separately.

4.2.1 Common Definitions

A system is composed by a set Π of n distributed processes $p_1, \ldots p_n$ interconnected by a point-to-point message system. Each process has its own local memory, and executes its own automaton. In every execution all processes stick to their specified automaton, except for f which prematurely halt. We call such processes crashed. Process that do not crash are correct as they take an infinite number of steps in infinite executions. All system models we consider assume an upper bound $t \geq f$ on the number of crashes in every execution.

A system model defines the behavior of the environment of the automatons with respect to a set of operations that bind together the automatons by allowing them to interact with each other by manipulating or querying their environment (e.g., the message system). Here, these operations are send and receive operations used to exchange messages from an alphabet $\mathcal{M}$. To simplify presentation, messages are assumed to be unique.

We define a partial run as an (infinite) sequence of global states C_i. A global state C_i is composed of the local states of the n automata corresponding to n processes and the message system (i.e., the messages in transit). We say that a process *takes a step*, when its local state and possibly also the state of the message system changes. A run is a partial run starting in an initial configuration.

A run is said to be admissible in a system model if the run sticks to the relations between the operations that are defined in the system model. As an example, all models here define the message system to be composed of reliable channels. Without going into the particular definitions of the send and receive operations, we can describe the abstract notion of reliable channels by the following three properties[2]:

Reliability. Every message sent to a correct process is eventually delivered.

Integrity. A message is delivered only if it was actually sent.

No Duplication. No message is received more than once.

Below (in Subsections 4.2.2, 4.2.3, 4.2.4, and 4.2.5) we will complement these definitions with additional assumptions. Since we are interested in the spatial distribution of synchrony here, these assumptions will only hold for some parts of the system. In fact, we will define families of synchrony models, which only differ in the size of these subsets. Note that (in contrast to [20]) the subset for which the synchrony holds is not known. Just the smallest size of this subset is known.

[2]Note that we did not choose the buffer representation as in [48] but used equivalent separate properties instead to characterize reliable channels.

4.2.2 The Failure Detector Model

In an asynchronous system model with a failure detector, a process p that executes a *well-formed algorithm* may execute during every computational step the following operations *in the given order*:

a-receive$_{\boldsymbol{p}}$**():** Delivers a message m, $\langle m, q\rangle \in \mathcal{M} \times \Pi$, sent from q to p.

a-query$_{\boldsymbol{p}}$**():** Queries the failure detector of p.

a-send$_{\boldsymbol{p}}\boldsymbol{(m, q)}$**:** Sends a message m to process q.

Note that all of these operations (but at least one) have to be performed in each step. Algorithms for the asynchronous model do not have access to global time.

A failure detector is of class $\mathcal{G}_k$, if it outputs a set of processes, k is a (possibly constant) function of the failure pattern, and the failure detector fulfills:

***k*-Eventual Trust.** In every execution, there exists a set Π' consisting of at least k of correct processes, such that there exists a time τ from which on the failure detector output of all correct processes is a set of correct processes and a superset of Π'.

The minimal instant τ is called *stabilization time*. Although this failure detector might seem artificial at first sight, it turns out to unify most of the classical stabilizing failure detectors in literature:

- The eventual strong failure detector $\diamond\mathcal{S}$ [28] guarantees that eventually at least one correct process is not suspected by any correct process. Therefore its output is the converse of the output of $\mathcal{G}_1$. That is, 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 any process' $\diamond\mathcal{S}$ is the (at least) one process that is in the intersection of the failure detector outputs of all correct processes.
- The eventual perfect failure detector $\diamond\mathcal{P}$ is the converse of $\mathcal{G}_{n-f}$.
- Finally, the eventual leader election oracle Ω [27] 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.

In the following, we will denote the family of asynchronous systems augmented with $\mathcal{G}_k$ for some k by Async+$\mathcal{G}$.

4.2.3 The Partially Synchronous Model

This section's model is a variant of the generalized partially synchronous model given in [28]. Based on the steps in a run, we define a discrete global timebase with instants $\tau \in \{0, 1, \ldots\}$, which is inaccessible to processes. At every instant of this time, every process may execute at most one step and at least one process executes a step. A process p that runs a *well-formed algorithm* executes at most one step at every discrete time τ and uses *one* of the following operations in every step:

par-send$_{\boldsymbol{p}}\boldsymbol{(m, q)}$**:** Sends a message m to q.

par-receive$_{\boldsymbol{p}}$**():** Delivers a set S, s.t. $\emptyset \subseteq S \subseteq M \times \Pi$ of message-sender pairs.

Definition 4.1. *We say that* Δ holds between p and q *at time* τ *provided that, if a message* m *is sent to* p *by* q *at time* τ *and* p *performs par-receive*$_q()$ *at time* $\tau' \geq \tau + \Delta$, m *is delivered to* p *by time* τ'.

Definition 4.2. *We say that* Φ *holds for* p *at time* τ*, when in the set of* $\Phi + 1$ *consecutive time instants starting with* τ *process* p *takes at least one step.*

In contrast to [48], Δ only holds for all *outgoing links* of k processes, which are called sources. We thus assume that in every execution there is a set of processes Π' of cardinality at least k such that:

***k*-Partial Sources.** Eventually some unknown Δ holds for all outgoing links of processes in Π'.

***k*-Partially Synchronous Processes.** Eventually some unknown Φ holds for each process in Π'.

The minimal time from which on the two properties hold for Π' is called stabilization time. (In contrast to [48], this stabilization time is not global, as it only holds for a subset of the system.)

We denote the system model where all executions fulfill *Reliability*, *Integrity*, *No Duplication*, *k-Partial Sources*, and *k-Partially Synchronous Processes*, as $\textsc{ParSync}_k$. If we do not fix k, we denote this family of models as $\textsc{ParSync}$.

Remark. Often a variant of partial synchrony is considered where message loss before the global stabilization time may occur. Here we consider only the case with *reliable links*, for the following reason: In [1] it is shown that fair lossy links can be transformed into reliable ones, if $n > 2t$, and that it is impossible to transform eventually reliable links into reliable links if $n \leq 2t$. So in the former case, which is also the relevant one for consensus, our results regarding solvability hold as well, whereas for the latter case, the opposite of our result is trivially true: It is not possible to build an asynchronous system with reliable links plus a failure detector in a partially synchronous system with message loss before the stabilization time.

Note that Dwork et al. [48] define another variant for partially synchronous communication, where Δ holds always, but is unknown. Since we have reliable channels, this is equivalent to our definition.

4.2.4 The Eventually Synchronous Model

The eventually synchronous model is a variant of partial synchrony, where the bounds on the communication delay and relative speeds are known, but hold only eventually. This is one of the two models in [48].

This model is very similar to the model of partial synchrony, for sake of brevity we do not go into much detail and only state the operations and properties of the model:

ev-send$_p(m, q)$**:** Sends a message m to q.

ev-receive$_p()$**:** Delivers a set S, s.t. $\emptyset \subseteq S \subseteq M \times \Pi$ of message-sender pairs.

As in the partially synchronous case, we consider a set of processes Π' of cardinality at least k, and we assume the following two properties:

***k*-Eventual Sources.** A known Δ eventually holds for all outgoing links of the processes in Π'.

***k*-Eventually Synchronous Processes.** A known Φ eventually holds for each process in Π'.

We denote the system model where all executions fulfill *Reliability*, *Integrity*, *No Duplication*, *k-Eventual Sources* and *k-Eventually Synchronous Processes*, as $\diamond\textsc{Sync}_k$. If we do not fix k, we denote this family of models as $\diamond\textsc{Sync}$.

Observe that $\textsc{ParSync}_k$ is — by definition — *not stronger* than $\diamond\textsc{Sync}_k$, that is, for any k, every execution in $\diamond\textsc{Sync}_k$ is also an execution in $\textsc{ParSync}_k$.

In order to distinguish the Δ for $\textsc{ParSync}$ and for $\diamond\textsc{Sync}$, we will use $\Delta_?$ for the former, and $\Delta_\diamond$ for the latter. When no ambiguity arises we will however only use Δ.

4.2.5 The Round Model

In our round-based system, processes proceed in rounds $r = 0, 1, 2, \ldots$. For a *well-formed algorithm*, in every round r, a process p executes exactly one step comprising a send operation followed by exactly one step comprising a receive operation, where the operations are defined as:

***rd-send*$_p(r, S)$:** Sends a set $S \subseteq M \times \Pi$ of messages. For every process q, S contains at most one message m_q.

***rd-receive*$_p(r)$:** Delivers a set $S \subseteq M \times \Pi \times \{0, 1, \ldots\}$ of messages to p, where a tuple $\langle m, q, r'\rangle$ denotes a message m sent by q to p in round $r' \leq r$.

Further, we define the property:

***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 the system model where all executions fulfill *Reliability*, *Integrity*, *No Duplication* and *k-Eventual Round Sources*, as $\textsc{Round}_k$. The family of all these models is denoted $\textsc{Round}$.

In our definition of *rd-receive* above we do not allow the reception of messages from future rounds. This implies that for each round r the *rd-receive*(r) operations form a consistent cut. By [89, Theorem 2.4] this is equivalent to these operations taking place in lockstep. Note also, that our model is communication open, and thus contrasts the communication closed round models used for example in [109, 30].

4.2.6 Algorithm Transformations

In order to relate our models, we use *algorithm transformations.* An algorithm transformation $\mathcal{T}_{A \rightharpoonup B}$ generates from *any* algorithm $\mathcal{A}$ that is correct (with respect to some problem specification) in some system $\mathcal{S}_A$ an algorithm $\mathcal{B}$ that is correct (with respect to the same specification) in some other model $\mathcal{B}$ by implementing the operations in $\mathcal{S}_A$ by operations in $\mathcal{S}_B$. For the correctness of such transformations, it has to be shown that $\mathcal{B}$ is well-formed as well, and that the implemented operations are those defined in $\mathcal{S}_A$. Moreover it has to be shown that the assumptions on the operations of $\mathcal{S}_A$ that $\mathcal{A}$ is based on, hold for their implementations (given by the transformation) in $\mathcal{S}_B$. This is captured by the notion of a trace: The *trace of the* $\mathcal{S}_A$ *operations* is the sequence of implementations of $\mathcal{S}_A$ operations (in $\mathcal{S}_B$) the algorithm calls when being executed via the transformation $\mathcal{T}_{A \rightharpoonup B}$ in $\mathcal{S}_B$.

Obviously, problem statements only make sense here if they can be stated independently of the model. Consequently, defining e.g. termination as "the consensus algorithm terminates after x rounds" is not model independent as there is no formal notion of "a round" e.g. in partially synchronous system models. Therefore the notion of a round in such models depends on the algorithms which implement them. Hence, such properties should be regarded as belonging to the algorithm and not to the problem and will be dealt with in the discussions on efficiency of transformations below. In the literature on models with stabilizing properties, algorithms which decide (or terminate) within an a priori bounded number of steps after stabilization time are termed *efficient*. Therefore we are interested in the possibility of transforming efficient algorithms into efficient algorithms.

An algorithm transformation is *B-bounded*, iff any *step* of the higher model can be implemented by at most B *operations* of the lower model.

Another measure for the efficiency of a transformation is how it behaves with respect to the stabilization in the two models involved. To motivate this measure, consider some implementation of e.g. the eventual perfect failure detector based on some partially synchronous system which has some global stabilization time τ. Since the timing before τ is arbitrary, the processes that some process p suspects may be arbitrary at τ as well. It may take some time until the set of suspected processes at p is consistent. Until then, the asynchronous algorithm using the failure detector may query the failure detector a couple of times — say x times — before the failure detector becomes consistent. Informally, x may be used as measure for the transformation of stabilizing properties.

More formally, since we are considering models with stabilizing properties there is usually a step s from which on $\mathcal{S}_B$ guarantees some properties. This step s is part of some step S in $\mathcal{S}_A$. For a valid transformation, there must also be a step S' from which on the stabilizing properties of the implemented model are guaranteed to hold. When $S \neq S'$ 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 after S.

Definition 4.3. *A transformation is* efficiency preserving*, if there are two a priori known values B and D such that the transformation is B-bounded and it has D-bounded shift.*

Note that this definition implies that parameters unknown in advance, e.g., $\Delta_?$ and $\Phi_?$ in the ParSync models, cannot occur in the expressions given for B, while the size of the system, n, can.

4.3 Equivalence of Solvability

4.3.1 Possibility of Async+$\mathcal{G}$$\rightharpoonup$ParSync

We begin by considering transformations of ASYNC+$\mathcal{G}$ algorithms to PARSYNC algorithms.

As there are no assumptions on the relative processor speeds and the time it takes for the network to transmit a message, all we need to show is that there is a set of processes that eventually meets the requirements of k-Eventual Trust.

Lemma 4.4. *In any* PARSYNC *run of an arbitrary algorithm $\mathcal{A}$ for an* ASYNC+$\mathcal{G}_k$ *system in conjunction with the transformation of Algorithm in Figure 4.1, the property k-Eventual Trust holds for the trace of the* ASYNC+$\mathcal{G}$ *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 Φ time steps, and messages originating from these processes arrive after Δ 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 $T' = (n+1)\Phi + \Delta$ steps between two consecutive messages 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 $T = T' + n$ steps after receiving m, for two consecutive messages m, m' sent to q from p.

Since *threshold*$_p$ is ever increasing, $\exists\tau, \forall\tau' \geq \tau : \textit{threshold}_p(\tau') > T$. Therefore, for all $p \in \Pi$ and for all $q \in \Pi'$ some message from q will be in *buffer*$_p$, when executing line 33. 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. □

All messages that are received via *par-receive* are stored into *buffer*$_p$ and then appended to *undelivered*$_p$, from where *a-receive* takes them while filtering out the additional $\perp$ messages, therefore our reliable channel assumptions follow from their counterparts of the PARSYNC-steps, and we obtain that:

Lemma 4.5. *The transformation of Algorithm in Figure 4.1 preserves* Reliability, Integrity *and* No Duplication.

Note that the ever increasing *threshold*$_p$ affects the detection time of the failure detector and has no impact on whether the transformation is bounded. However, for the ASYNC+$\mathcal{G}$-algorithm stabilization only occurs after *threshold*$_p$ is greater than $\Delta_? + (n+1)\Phi_?$ (cf. Section 4.4.2). As $\Delta_?$ and $\Phi_?$ are unknown there is no a priori known bound for the stabilization shift.

Corollary 4.6. *Algorithm in Figure 4.1 transforms any algorithm $\mathcal{A}$ for* ASYNC+$\mathcal{G}_x$ *to an algorithm for* PARSYNC$_x$*, and this transformation is not efficiency preserving.*

4.3.2 Possibility of ParSync$\rightharpoonup$$\diamond$Sync

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

Algorithm 1 Transforming ASYNC+$\mathcal{G}$ algorithms to PARSYNC algorithms

```
variables
    $\forall q \in \Pi : \; send_p[q]$, initially $\bot$
    $trusted_p \subseteq \Pi$, initially $\emptyset$
    $buffer_p, \; undelivered_p$ FIFO queue with elements in $\mathcal{M} \times \Pi$, initially empty
    $counter_p, \; threshold_p \in \mathbb{N}$, initially 0 and 1, resp.
end variables

operation $a\text{-}send_p(m, q)$
    $send_p[q] \leftarrow m$
    $update()$
end operation

operation $a\text{-}receive_p()$
    $update()$
    if $undelivered_p$ does not contain non-$\bot$ messages then
        return $\bot$
    else
        return first $\langle m, q \rangle$ from $undelivered_p$ where $m \neq \bot$
            [removing all pairs up to and including $\langle m, q \rangle$]
    end if
end operation

operation $a\text{-}query_p()$
    $update()$
    return $trusted_p$
end operation

procedure $update()$
    for all $q \in \Pi$ do
        $par\text{-}send(send_p[q], q)$
        $send_p[q] \leftarrow \bot$
    end for
    append all $\langle m, q \rangle \in par\text{-}receive_p()$ to $buffer_p$
    $incr(counter_p)$
    if $counter_p = threshold_p$ then
        $counter_p \leftarrow 0$
        $trusted_p \leftarrow \{q : \langle *, q \rangle \in buffer_p\}$
        append $buffer_p$ to $undelivered_p$
        $buffer_p \leftarrow \emptyset$
        $incr(threshold_p)$
    end if
end procedure
```

Figure 4.1: Transforming ASYNC+$\mathcal{G}$ algorithms to PARSYNC algorithms.

Corollary 4.7. *There exists a 1-bounded transformation for a* ParSync *algorithm to a* $\diamond$Sync *algorithm, without stabilization shift.*

4.3.3 Possibility of $\diamond$Sync$\rightharpoonup$Round

Now we transform any algorithm for $\diamond$Sync to an algorithm for the Round model.

While each $\diamond\text{Sync}_k$ model can be instantiated with any values for $\Delta_\diamond$ and $\Phi_\diamond$, we implement a particular instance, i.e., $\diamond\text{Sync}_k$ with $\Delta_\diamond = 0$ and $\Phi_\diamond = 1$. The basic idea of the transformation is to execute one round of the Round model in each $\diamond$Sync step.

Lemma 4.8. *In any* Round *run of an arbitrary algorithm $\mathcal{A}$ for an* $\diamond$Sync *system in conjunction with Algorithm 2 in Figure 4.2, the properties* k-Eventual Sources *and* k-Eventually Synchronous Processes *with $\Phi = 1$ and $\Delta = 0$ hold for the trace of* $\diamond$Sync *operations.*

Proof. To establish the lemma we show that in the trace of $\diamond$Sync operations the k eventual source(s) of Round_k will form a set of processes denoted $\Pi' \subseteq \Pi$, with $|\Pi| = k$, for which $\Phi = 1$ and $\Delta = 0$ eventually hold. That is, we show that there is a one-to-one mapping of the instances of *ev-send* and *ev-receive* operations to lock step rounds — i.e., each $\diamond$Sync operation corresponds to two Round steps. Eventually, if a message m is sent by a source via an *ev-send* corresponding to round r at the sender, the receiver will put it into $\textit{buffer}_p$ (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 Definition 4.1 we get $\Delta = 0$.

To see why eventually Φ holds for the processes in Π', we observe that (as noted above) the *rd-receive* steps induce an (infinite) sequence of consistent cuts, which we can use as a "relativistic" timebase. In this timebase, each process in Π' takes exactly one $\diamond$Sync-step at each point in time. Therefore we have $\Phi = 1$. □

Lemma 4.9. *Algorithm 2 in Figure 4.2 has no stabilization shift.*

Proof. Let r be the first round where k-Eventual Round Sources holds, and let s denote the $\diamond$Sync-step in which implementation round r occurs. Then for all rounds $r' > r$ and all step s' corresponding to r', all messages sent by processes belonging to Π' in round r' arrive in round r', s.t., they are available for delivery by the first *ev-receive*-step following r', and (according to the proof of Lemma 4.8 above) all processes take a step at the "relativistic" time corresponding to round r'. □

Lemma 4.10. *Algorithm 2 in Figure 4.2 preserves* Reliability, Integrity *and* No Duplication.

It can be easily seen that we have:

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

yielding that this transformation is 2-*bounded*, since only one of the two operations is possible in each step of our delay-bounded models. Moreover, from Lemmas 4.8, 4.9 and 4.10 it is obvious that:

Corollary 4.11. *Algorithm 2 in Figure 4.2 transforms any algorithm $\mathcal{A}$ for* $\diamond\text{Sync}_x$ *to an algorithm for* Round_x*, and this transformation is 2-bounded and does not shift stabilization.*

Algorithm 2 Transforming $\diamond$SYNC algorithms to ROUND algorithms.

```
variables
    $r \in \mathbb{N}$, initially 0
    $buffer_p \subseteq \mathbb{N} \times \mathcal{M} \times \Pi$, initially $\epsilon$
end variables

operation $ev\text{-}send_p(m, q)$
    $rd\text{-}send_p(r, \{\langle m, q \rangle\})$
    $buffer_p \leftarrow buffer_p \cup rd\text{-}receive_p(r)$
    $r \leftarrow r + 1$
end operation

operation $ev\text{-}receive_p()$
    $rd\text{-}send_p(r, \emptyset)$
    $buffer_p \leftarrow buffer_p \cup rd\text{-}receive_p(r)$
    $r \leftarrow r + 1$
    $del \leftarrow \{\langle m, q \rangle \mid \langle m, q, * \rangle \in buffer_p\}$
    $buffer_p \leftarrow \emptyset$
    return $del$
end operation
```

Algorithm 3 Transforming ROUND algorithms to ASYNC+$\mathcal{G}$ algorithms.

```
variables
    $undelivered_p \subseteq \mathbb{N} \times \mathcal{M} \times \Pi$, initially empty
end variables

operation $rd\text{-}send_p(S, r)$
    for all $\langle m, q \rangle \in S$ do
        $a\text{-}send_p(\langle r, m \rangle, q)$
    end for
    for all $q \notin \{q' | \langle m', q' \rangle \in S\}$ do
        $a\text{-}send_p(\langle r, \perp \rangle, q)$
    end for
end operation

operation $rd\text{-}receive_p(r)$
    repeat
        $undelivered_p \leftarrow undelivered_p \cup \{a\text{-}receive_p()\}$
        $trusted_p \leftarrow a\text{-}query_p()$
        $Q \leftarrow \{q \mid \langle r, *, q \rangle \in undelivered_p\}$
    until $Q \supseteq trusted_p$
    $del \leftarrow \{\langle r', m, q \rangle \in undelivered_p \mid r' \leq r \wedge m \neq \perp\}$
    $undelivered_p \leftarrow undelivered_p \setminus del$
    return $del$
end operation
```

Figure 4.2: Transforming $\diamond$**Sync** algorithms to **Round** algorithms (Algorithm 2) and Transforming **Round** algorithms to **Async**+$\mathcal{G}$ algorithms (Algorithm 3).

4.3.4 Possibility of Round$\rightharpoonup$Async+$\mathcal{G}$

With this section's transformation, we close the circle of transformations thereby establishing that the same problems are solvable in all four model families.

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 trusted processes.

Lemma 4.12. *In any* Async+$\mathcal{G}$ *run of an arbitrary algorithm* $\mathcal{A}$ *for a* Round *system in conjunction with the transformation of Algorithm 3 in Figure 4.2, 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., $\langle *, \perp\rangle$) 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 17).

For each q which is eventually trusted forever, there is a round r for which it is trusted by all processes when executing line 18. Obviously from now on (i.e., for rounds $r' \geq r$) a message from q must be contained in the $\mathit{undelivered}_q$ set of every alive process every time a process reaches this line. If the content of the message is not $\perp$ it will also be delivered, thereby ensuring that every correct process does indeed receive p's round r' message (if it sent one). ☐

Lemma 4.13. *Algorithm 3 in Figure 4.2 preserves* Reliability*,* Integrity *and* No Duplication*.*

Again, this lemma follows directly from the transformation and from the fact that the properties are provided by Round (cf. for example lines 14 and 19 for *Integrity* and *No Duplication*, resp.). Transforming Round to Async+$\mathcal{G}$ with Algorithm 3 in Figure 4.2, we have:

$$\mathit{rd\text{-}send} \mapsto \begin{cases} \mathit{a\text{-}send}(*, 1) \\ \vdots \\ \mathit{a\text{-}send}(*, n) \end{cases} \qquad \mathit{rd\text{-}receive} \mapsto \begin{cases} \mathit{a\text{-}receive} \\ \mathit{a\text{-}query} \\ \mathit{a\text{-}receive} \\ \mathit{a\text{-}query} \\ \vdots \end{cases}$$

yielding that this transformation is *unbounded*. (Although the *a-send* are not necessarily executed in the given order.) Therefore we conclude this Subsection with observing that:

Corollary 4.14. *Algorithm 3 in Figure 4.2 transforms any algorithm* $\mathcal{A}$ *for* Round$_k$ *to an algorithm for* Async+$\mathcal{G}_k$*, and this transformation is not efficiency preserving.*

4.4 Efficiency of Transformations

In the previous section we showed that from a solvability point of view, all four models are equivalent. The chain of transformations of a ParSync algorithm to a $\diamond$Sync algorithm to a Round algorithm can be done with bounded transformation, i.e., the transformation is *efficiency preserving.* That means, e.g., if there is already an efficient algorithm for the ParSync model, the transformed algorithm is also efficient in the $\diamond$Sync and the Round model. On the other hand, in this section we show that there is no transformation that maintains this efficiency for (1) the other transformations in the previous section and (2) for the transformations going backwards in the efficient chain. 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.

4.4.1 Lower Bounds

The aim of this section is to show that which transformations cannot be efficiency preserving. The (trivial) idea behind the proof that no transformation ParSync$\rightharpoonup$Async+$\mathcal{G}$ can be efficiency preservingis that no message with an unbounded delay can be received in a bounded number of steps.

Theorem 4.15. *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 T that is B-bounded for some B. By the definition of ParSync, there exists a Δ, a time τ_{stab}, and a set of processes C, with $|C| = k$, so that all messages sent by a $p \in C$ at $\tau \geq \tau_{stab}$ are received by ParSync time $\tau + \Delta$ at q. Assuming that the sending of such a message m at ParSync time τ happens at Async+$\mathcal{G}$ time τ', the transformation has to ensure that it can deliver m in the *par-receive* step following the Async+$\mathcal{G}$ time $\tau' + B\Delta$. However, in Async+$\mathcal{G}$ there is no guarantee that m is received by q by this time, and thus arbitrarily (finitely) many asynchronous steps might be necessary. Thus q is not able to deliver this message after a constant number of steps. □

We can also show that no transformation in the opposite direction can be efficiency preserving. The main idea is that no reliable suspicion of faulty processes can be made within known time in a system with unknown delays.

Theorem 4.16. *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 are going to show that any transformation from Async+$\mathcal{G}$ to ParSync is either unbounded or has unbounded shift.

However, this is easy to see, since if a transformation is bounded with bounded shift, this implies that every process stops suspecting all sources after a bounded number of steps after stabilization. Let L be such a bound, and τ_L the earliest time where all correct processes made L steps after stabilization. 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 correct processes arrives before τ_L. Since Δ is unknown, and τ_L lies only bounded time after stabilization, such a run might exist. Let Q be the set of processes that are trusted by a correct process at τ_L in R. As already mentioned, we must have $|Q| \geq k$. Now pick one of these processes, say

q, and consider a run R' that is similar to R, except that q has crashed initially. Since R' is indistinguishable from R, q will be trusted by all processes, a contradiction to the assumption that only correct processes are trusted at this time. $\square$

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

Theorem 4.17. *There exists no efficiency-preserving transformation that transforms any algorithm $\mathcal{A}$ for $\diamond$Sync$_k$ to an algorithm for* ParSync$_k$.

Proof. Assume by contradiction that there exists a transformation T that is B-bounded for some known B. By the definition of $\diamond$Sync, there exists a *known* $\Delta_\diamond$, a time $\tau_\diamond^0$, and a set of processes C, with $|C| \geq k$, so that all messages sent by some $p \in C$ at $\tau_\diamond \geq \tau_\diamond^0$ are received by $\diamond$Sync time $\tau_\diamond + \Delta_\diamond$ at q. Assuming that the sending of such a message m at $\diamond$Sync time $\tau_\diamond$ happens at ParSync time $\tau_?$, the transformation has to ensure that it can deliver m in the *ev-receive*-step following the ParSync time $\tau_? + B\Delta_\diamond$. However, in ParSync, m is guaranteed to be received by q only after some $\Delta_?$, that is *unknown* a priori, and may be greater than $B\Delta_\diamond$ in some runs. Thus q is not able to deliver this message after an a priori known number of steps in all runs. $\square$

Our next lower bound follows from Theorem 4.15 and Corollary 4.7: If there was an efficiency-preserving transformation $\diamond$Sync$\rightharpoonup$Async+$\mathcal{G}$ these two results would be contradictory.

Theorem 4.18. *There exists no efficiency-preserving transformation that transforms any algorithm $\mathcal{A}$ for $\diamond$Sync to an algorithm for* Async+$\mathcal{G}$.

Proof. Follows from Theorem 4.15 and Corollary 4.7: If such an efficiency-preserving transformation existed, the combination of this transformation with the (trivial) transformation of Corollary 4.7 would contradict Theorem 4.15. $\square$

To prove that the transformations considered above are necessarily not efficiency preserving it was sufficient to examine only stabilization shift. Conversely, our proof there is no efficiency-preserving transformation Round$\rightharpoonup\diamond$Sync is based on showing that no transformation can be B-bounded and, at the same time, cause only D-bounded shift.

Theorem 4.19. *There exists no efficiency-preserving transformation that transforms any algorithm $\mathcal{A}$ for* Round$_k$ *to an algorithm for* $\diamond$Sync$_k$.

Proof. Assume by contradiction that an efficiency-preserving transformation exists. Then there must be two known bounds B and D, such that the transformation is B-bounded and causes D-bounded stabilization shift. The argument of the calls of the implementation of the operations of the Round model imply a round number at each step of the $\diamond$Sync model. During the unstable period the round numbers of two processes p and q may drift arbitrarily far from each other. (Because the transformation is B-bounded but Φ does not hold yet.) Now assume w.l.o.g. that p is the process that has the lowest round number, but is one of the k processes for which Δ and Φ hold once the system stabilizes (at τ_{stab}), and q has the highest round number and is not in the set of k eventually synchronous processes. Then the transformation has to ensure that p will reach the highest round number in the system within D rounds (recall that by assumption the stabilization shift is bounded by

D), since p must become a ROUND source, i.e., must be heard by all processes within the same round. This, however, is impossible since it may take an unbounded number of $\diamond$SYNC-steps until p learns that q has a larger round number and starts to catch up. $\square$

4.4.2 Upper Bound on Async+$\mathcal{G}$$\rightharpoonup$$\diamond$Sync

We use a modified version of Algorithm in Figure 4.1, with the following changes: $\mathit{threshold}_p$ is initialized to $\Delta_\diamond + (n+1)\Phi_\diamond$ and the last line is omitted. This incorporates that we know $\Delta_\diamond$ and $\Phi_\diamond$ in advance, and thus we do not have to estimate it.

It is clear that the proof of Algorithm in Figure 4.1 analogously applies here and thus this transformation is correct. It is also easy to see that it is also $(n+1)$-bounded:

$$\textit{a-send}, \textit{a-receive}, \textit{a-query} \quad \mapsto \quad \begin{cases} \textit{ev-send}(*, 1) \\ \vdots \\ \textit{ev-send}(*, n) \\ \textit{ev-receive} \end{cases}$$

To determine the bound on the stabilization shift, note that since every source s will send a message to p in every ASYNC+$\mathcal{G}$ operation (taking at most $(n+1)\Phi$ $\diamond$SYNC steps) and messages are delivered within $\Delta_\diamond$ $\diamond$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 $\Delta_\diamond + (n+1)\Phi_\diamond$ $\diamond$SYNC steps, and consequently:

Theorem 4.20. *There exists an efficiency-preserving transformation that transforms any algorithm $\mathcal{A}$ for* ASYNC+$\mathcal{G}_k$ *to an algorithm for* $\diamond$SYNC$_k$ *.*

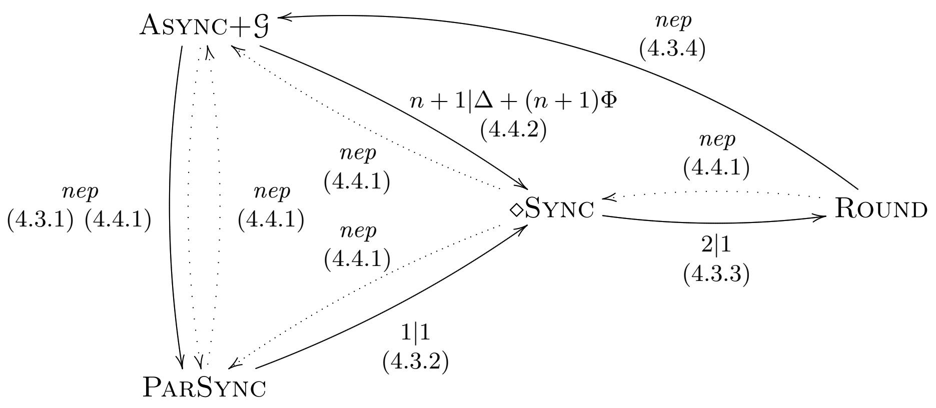

Figure 4.3: Relations of models with pointers to sections here; an arrow from model M_1 to model M_2 indicates that a result on algorithm transformations from model M_1 to model M_2 can be found here. Solid lines indicate upper bounds, dotted lines indicate lower bounds. $B|D$ means that a transformation exists which is B-bounded and has D-bounded shift. We use *nep* to mark non-efficiency-preserving transformations.

4.5 Discussion

Solvability. Figure 4.3 presents a graphical overview of our results. It can easily be seen that the directed subgraph consisting of solid arrows is strongly connected. This subgraph presents the transformation algorithms we have provided here. Thus, all model families presented are equivalent regarding solvability.

Relation to the results of [29]. Setting $k = n - f$ in our models, $\textsc{Round}_{n-f}$ is in fact the classic basic round model by Dwork et al. [48] and the asynchronous model with $\mathcal{G}_{n-f}$ is the asynchronous model augmented with the eventually perfect failure detector [28]. We use $k = n - f$ in order to intuitively discuss why these stabilizing models are equivalent from a solvability viewpoint while their perpetual counterparts are not.

The main observation in the relation to [29] is concerned with the term "eventually" in the model definition: In the asynchronous model augmented with $\diamond\mathcal{P}$, two things happen in every execution: (1) eventually, the failure detector becomes accurate, and (2) eventually, the last process crashes and all its messages are received.

The model considered in [29] (asynchrony with $\mathcal{P}$), however shares only (2) while the failure detector taken into account satisfies perpetual strong accuracy. It was shown in [29] that given (2), even perpetual strong accuracy — and thus $\mathcal{P}$ — is too weak to implement a model where every round is communication closed and reliable, as is required by the synchronous model of computation: 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. [58]).

For showing our equivalence result in the special case of $n - f$, one has to show that communication closed rounds are ensured eventually. We observe that (1) and (2) are sufficient to achieve this. After (1) and (2) hold, all processes are correct, they will never be suspected and thus all their messages are received in the round they were sent. Thus we achieve communication closed rounds eventually which is equivalent to eventual lock-step and thus eventual synchrony.

Timing Assumptions. Our results show the equivalence of diverse models with stabilizing properties in which 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 its messages are 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 [51] 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 in here.

Number of Timely Links. An interesting consequence of our results concerns models where only t links of the sources are eventually timely [4]. 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 $\textsc{ParSync}_1$ (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.

Efficiency. An algorithm solving some problem in a stabilizing model is efficient, if it decides (i.e., terminates) after a bounded number of steps after stabilization. Consider some efficient algorithm $\mathcal{A}$ working in model M. If there exists an efficiency-preserving transformation from M into some model M', this implicates that there exists an efficient algorithm in M' as well (the resulting algorithm when $\mathcal{A}$ is transformed). The converse, however, is not necessarily true. The dotted lines with a *nep* label in Figure 4.3 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 said about the existence of an efficient solution in the absence of an efficiency-preserving transformation from M to M'. As an example, consider $\textsc{Round}_{n-f}$ and $\diamond\textsc{Sync}_{n-f}$ for which efficient consensus algorithms were given in [46] and [47], respectively. Despite this fact, our results show that there cannot be a (general) efficiency-preserving transformation from Round to $\diamond$Sync algorithms.

Chapter 5

Randomized Message Omission Robust Protocols

In the *fair exchange* problem, mutually untrusting parties must securely exchange digital goods. A fair exchange protocol must ensure that no combination of cheating or failures will result in some goods being delivered but not others, and that all goods will be delivered in the absence of cheating and failures. An interesting feature of fair exchange is that it can be reduced to consensus when trusted coprocessors are available. Another one is that fair exchange not only may provide fairness in electronic commerce but also is in fact analogous to the problem of atomic commit in database repositories.

We begin our exposition of message omission robust protocols by proposing two novel randomized protocols for solving fair exchange using simple trusted coprocessors [54]. Both protocols have an optimal expected running time, completing in a constant (3) expected number of rounds. They also have optimal resilience. The first one tolerates any number of dishonest parties, as long as one is honest, while the second one, which assumes more agressive cheating and failures assumptions, tolerates up to a minority of dishonest parties.

The key insight is similar to the idea underlying the *code-division multiple access* (CDMA) communication protocol: outwitting an adversary is much easier if participants share a common, secret pseudo-random number generator.

The presentation is structured as follows. After introducing some motivation (Section 5.1), we describe the model of computation considered (Section 5.2), show how to reduce fair exchange to uniform consensus (Section 5.3), and display related work (Section 5.4). Optimal randomized uniform consensus protocols for binary inputs with a constant (3) expected number of rounds are then introduced (Section 5.5 and Section 5.6). Note that both protocols may be generalized to a larger set of k values with a factor of $\log(k)$. However, we concentrate on the binary case, as we are interested in solving fair exchange efficiently.

5.1 Motivation

In the *fair exchange* problem, a set of parties want to trade an item which they have for an item of another party (for a survey of fair exchange see [99]). Fair exchange is a fundamental problem in domains with electronic business transactions since (1) items can be any type of electronic asset (electronic money, documents, music files, etc.) and (2) fairness is especially important in rather anonymous environments without means to establish mutual trust relationships. Briefly spoken, fair exchange guarantees that (1)

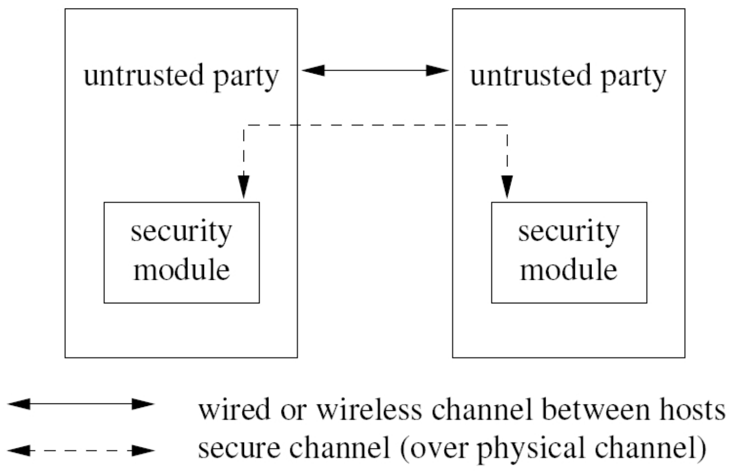
untrusted party
untrusted party
security module
security module
wired or wireless channel between hosts
secure channel (over physical channel)

Figure 5.1: Untrusted parties and security modules.

every honest party eventually either delivers its desired item or aborts the exchange, (2) the exchange is successful if no party misbehaves and all items match their descriptions, and (3) the exchange should be fair, i.e., if the desired item of any party does not match its description, then no party can obtain any (useful) information about any other item. Fair exchange algorithms must guarantee these properties even in the presence of arbitrary (malicious) misbehavior of a subset of participants.

Fair exchange, a security problem, can be reduced [11] to a fault-tolerance problem, namely a special form of *uniform consensus*. In the (non-uniform) consensus problem [101], each process in a group starts with a private input value, and after some communication, each non-faulty process is required to decide (termination) on the same private output value (agreement), so that all processes that decide choose some process's private input value (validity). In its uniform version, however, agreement requires all processes that decide (faulty or non-faulty) to decide the same value. Only non-faulty processes are required to terminate.

The reduction from fair exchange to consensus [11] holds in a synchronous model where each participating party is equipped with a trusted coprocessor, that is, a tamper-proof security module like a smart card (see Fig. 5.1). Security modules have recently been advocated by key players in industry to improve the security of computers in the context of *trusted computing* [114]. Today, products exist which implement such trusted devices (see for example [49]). Roughly speaking, a security module is a certified piece of hardware executing a well-known algorithm. Security modules can establish confidential and authenticated channels between each other. However, since they can only communicate by exchanging messages through their (untrusted) host parties, messages may be intercepted or dropped. Overall, the security modules form a *trusted subsystem* within the overall (untrusted) system. The integrity and confidentiality of the algorithm running in the trusted subsystem is protected by the shield of tamper proof hardware. The integrity and confidentiality of data sent across the network is protected by standard cryptographic protocols. These mechanisms reduce the type and nature of adversarial behavior in the trusted subsystem to message loss and process self-destruction, two standard fault-assumptions known under the names of *omission* and *crash* in the area of fault-tolerance.

Here we propose two novel randomized protocols for solving uniform consensus with

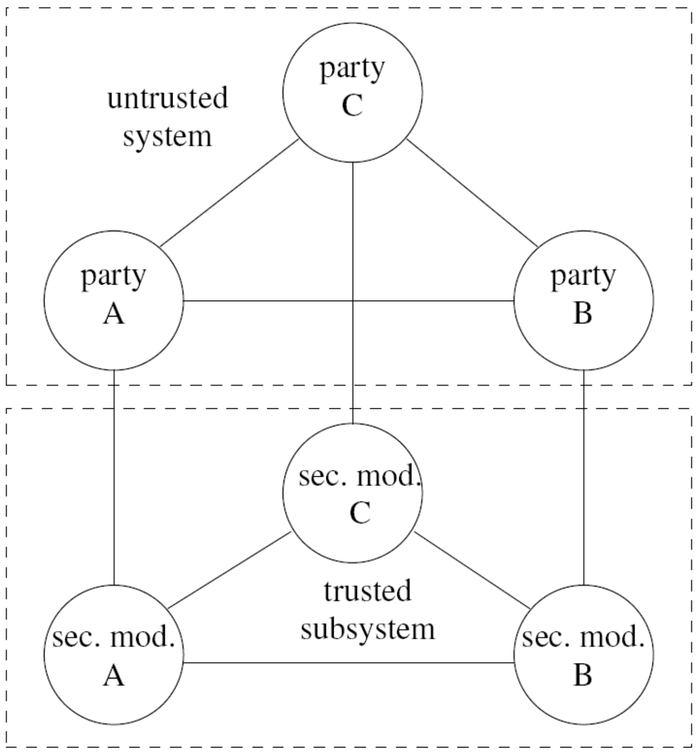

Figure 5.2: The untrusted system and the trusted subsystem.

binary inputs (and hence fair exchange) using such trusted coprocessors. Our protocols are time optimal, completing in a constant (3) expected number of rounds. They are also optimal in terms of resilience. The essential idea lies in participants sharing a common, secret pseudo-random number generator. In a multi-round protocol, each trusted coprocessor can flip a coin, and take action secure in the knowledge that every other trusted coprocessor has flipped the same value, and is taking a compatible action in that round. Because messages are encrypted, coin flip outcomes can be hidden, so dishonest parties can neither observe past coin flips nor predict future ones. (Of course, the pseudo-random algorithm itself need not be secret as long as the trusted coprocessors' common seed is kept secret, just like their common cryptographic key.) We believe that this approach is both efficient and practical.

5.2 Model of Computation

Our model of computation is essentially synchronous: participants exchange messages in synchronous rounds. Of course, real distributed systems are not synchronous in the classical sense, but it is reasonable to assume an upper bound on how long one can expect a non-faulty processor to take before responding to a message. A processor that takes too long to join in a round is assumed to be faulty or malicious.

The system is logically structured into an untrusted system (including the untrusted parties and their communication channels) and the trusted subsystem consisting of the parties' individual trusted coprocessors, that is, their tamper-proof security modules (see

Fig. 5.2). The untrusted parties can interact with their trusted coprocessors through a well-defined interface, but they cannot in any other form influence the computation within the trusted coprocessor.

As noted, communication among the trusted coprocessors is confidential and authenticated, so malicious parties cannot interpret or tamper with these messages. Because each trusted coprocessor sends the same encrypted message to every other trusted coprocessor, we have receiver anonymity and so a cheating party cannot learn who is communicating to who from traffic analysis. An untrusted party can, however, prevent outgoing messages from being sent (called a *send omission*), or incoming messages from being received (called a *receive omission*) or destroy its trusted coprocessor (called a *crash*). The effects of a crash can be regarded as a permanent send (and receive) omission.

Define a party as *cheating* if it causes send or receive omissions of its trusted coprocessor. A party which does not cheat is *honest*. A fair exchange protocol must ensure that under no circumstances will goods be delivered to a cheating party but not to all honest parties. It is, however, acceptable to deliver the goods to all honest parties, but not to some cheating parties. Cheating may cause the exchange to *fail*, so that no goods are delivered to any party. In the absence of cheating, the exchange should *succeed*, causing goods to be delivered to all participants. For brevity, we refer to processes when we really mean untrusted processes equipped with trusted coprocessors. With a *process failure* we mean either a crash, a send message omission or a receive message omission.

5.3 Fair Exchange as Consensus

The reduction from fair exchange to uniform consensus works as follows. In the first round of the protocol, each party applies its acceptance test to the encrypted digital goods received from the others (in special cases this test can also be performed within the trusted coprocessor). It then informs its trusted coprocessor whether the goods passed the test. The trusted coprocessors broadcast this choice (using confidential and authenticated messages) within the trusted subsystem. Each coprocessor that receives unanymous approvals starts the consensus protocol with input 1, and each trusted coprocessor that either observes a disapproval or no message from a trusted coprocessor starts the consensus protocol with input 0. At the end of the protocol, each trusted coprocessor delivers the goods if the outcome of the protocol is 1, and refuses to do so if the outcome is 0.

It is easy to see that in the absence of failures or cheating all goods will be delivered. The uniform consensus protocol ensures that all honest parties agree on whether to deliver the goods, and its uniformity ensures that no trusted coprocessor residing at a cheating party will deliver the goods if any trusted coprocessor at an honest party decides not to. (Recall that it is acceptable if the honest processes deliver the goods after deciding 1, even if a cheating process fails to deliver the goods after deciding 0.)

As noted, the protocols considered here are *randomized*, in the sense that they rely on the assumption that trusted coprocessors generate pseudo-random values that cannot be predicted by an adversary. These protocols always produce correct results, but their running time is a random variable, the so-called *Las Vegas* model. (It is straightforward to transform these protocols into *Monte-Carlo* protocols that run for a fixed number of rounds, and produce correct results with very high probability.)

To simplify the presentation, we first present an uniform consensus protocol that works in a failure model that permits send, but not receive omissions. This protocol is slightly simpler and more robust: it tolerates $f < n$ cheating processes, while the full send/receive omissions model protocol tolerates $f < n/2$ failures. Both resilience levels are optimal for their respective models [100]. Presenting the protocol in two stages illustrates how assumptions about the model affect the protocol's complexity and resilience.

5.4 Related Work

We build on work of Parvédy and Raynal [100]. They derive optimal early stopping deterministic uniform consensus algorithms for synchronous systems with send or send/receive omission failures. However, our algorithms are more efficient in most cases (if the number of failures is not constant) and at least comparable (otherwise).

Feldman and Micali [50] exhibit optimal consensus algorithms for Byzantine agreement, which in principle could also be used in omission failure models. Despite having also an optimal expected running time, our algorithms outperform theirs both on resilience and on the probability of not having termination violated.

Avoine, Gärtner, Guerraoui and Vukolic [11] show how to reduce the fair exchange problem in a system where processes are provided with security modules to the consensus problem in omission failure models. A solution to the fair exchange problem is presented by use of the algorithms of Parvédy and Raynal [100]. In the same context, Delporte, Fauconnier and Freiling [38] investigate solutions to consensus for *asynchronous* systems which are equipped with unreliable failure detectors. They exhibit a weak failure detector in the spirit of previous work by Chandra, Hadzilacos and Toueg [27] that allows to solve asynchronous consensus in omission failure environments.

Aspnes [7] presents a survey of randomized consensus algorithms for the shared memory model where processes are prone to crashes. These results are particularly interesting, since consensus cannot be solved deterministically in a pure asynchronous distributed system, as proved in [52] by Fischer, Lynch and Paterson.

5.5 Optimal Protocol for Send Omissions

The *ConsensusS* algorithm in Figure 5.3 solves uniform consensus with binary inputs in optimal 3 expected synchronous rounds tolerating an optimal number of up to $f < n$ failures - send message omissions as well as process crashes.

As noted, all processes share a common secret seed and pseudo-random number generator. We denote the r-th such pseudo-random binary number by $\mathit{flip}(r)$. For each r, every process computes the same value for $\mathit{flip}(r)$.

In ConsensusS, each process broadcasts its binary input (line 1). In each subsequent round, the process waits to hear each process's *preference.* If they disagree (line 3), the process broadcasts a message informing the others. When receiving such a broadcast for the first time (line 6), every process relays it. Hence, if any non-faulty process receives mixed preferences or a $\mathit{disagreement}(r)$ message, then all processes receive a $\mathit{disagreement}(r)$ message and will change preference according to the coin flip. If they agree (line 9) and no message communicating disagreement seen by another process is received, then the process checks whether that preference agrees with the common pseudo-random binary number for that round. If so, it is safe to decide that value (line 11). If not, the process simply rebroadcasts the preference (line 13). If the preferences disagree or the process is informed so, then the process uses the common pseudo-random binary number to choose a new preference (line 16). If any process announces that it has decided, then the process decides on the same value (line 18).

Very informally, this protocol exploits in an essential way the observation that each process (but not the adversary) can predict the others' next coin flips. If a process receives

```
send prefer(binary preference) to all;
for each round r {
  if (both prefer(0) and prefer(1) received) {
    send disagreement(r) to all;
  }
  on (receipt of disagreement(r) for the first time) {
    send disagreement(r) to all;
  }
  if (all received preferences are prefer(v)) and (no disagreement(r) received){
    if (flip(r) == v) {
      send decide(v) to all and return(v);
    } else {
      send prefer(v) to all;
    }
  } else {
    send prefer(flip(r)) to all;
  }
  if (any decide(v) received) {
    return(v)
  }
}
```

Figure 5.3: Uniform consensus for send message omissions and process crashes.

v from all processes, then v was sent by at least one good process, so every other process will either receive all v preferences or both preferences. Any processes that receive either mixed preferences or *disagreement*(r) messages will change preference according to the coin flip. If the coin flip is the same as v, then all processes will prefer v, and it is safe to decide.

Lemma 5.1. *If $f < n$, for every process the expected number of rounds of ConsensusS is 3, and the protocol terminates with probability 1.*

Proof. Think of an execution as a tree, where the root node represents the initial round and the children of a node represent the following round possibilities. Let $E(n)$ be the expected number of rounds from node n. If n has children $n.1$ and $n.2$, chosen by coin flip, then $E(n) = (1/2)(1 + E(n.1)) + (1/2)(1 + E(n.2))$. Each child contributes one plus its expected running time, but with probability one-half. Now let

- $E(n) = E_1(n)$ if at node n some non-faulty processes sent *prefer*(0) and some non-faulty processes sent *prefer*(1),
- $E(n) = E_2(n)$ if at node n all non-faulty processes sent *prefer*(v) and some non-faulty processes receive a disagreement message or both *prefer*(0) and *prefer*(1),
- $E(n) = E_3(n)$ if at node n all non-faulty processes sent *prefer*(v) and all non-faulty processes receive no disagreement messages and only *prefer*(v).

Note that if $E(n) = E_z(n)$ and $E(n.1) = E_w(n.1)$, it may be that $z \neq w$. However, it is always the case that if $E(n.1) = E_z(n.1)$ then $E(n.2) = E_z(n.2)$. The reason is that from one round to the other the values that the non-faulty processes send and receive may change. However, if the non-faulty processes behave in a way at one children, then they should behave the same way at the other, since both children just differ in the coin flip. Hence, executions differing themselves by the values sent and received by non-faulty processes may generate distinct execution trees.

Now let e be the root of an execution tree. Consider that

- $E(e) = E_1(e)$: If there are non-faulty processes that sent *prefer*(0) and other non-faulty processes that sent *prefer*(1) in round r, then at round $r + 1$ every process receives at least one message *prefer*(0) and one message *prefer*(1), and thus, from round $r + 1$ on, all preference messages sent by every process (and all received as well) will be *prefer*(*flip*($r + 1$)). Hence, all processes will decide on *flip*($r + 1$) in the first round t such that *flip*(t) $= flip(r + 1)$, and the probability that any process (and thus, a non-faulty one) violates termination is the same as the probability that such a round t never happens, that is, zero. Besides, the expected number of rounds to achieve a round t such that *flip*(t) $= flip(r + 1)$ is 2. Thus, the expected number of rounds of ConsensusS is $3 = E_1(e) = (1/2)(1 + 2) + (1/2)(1 + 2)$.

- $E(e) = E_2(e)$: If all non-faulty processes sent *prefer*(v) in round r and part of the non-faulty processes receive a disagreement message or both messages *prefer*(0) and *prefer*(1) in round $r + 1$, then all processes receive disagreement messages and from round $r + 1$ on, all preference messages sent by every process (and all received as well) will be *prefer*(*flip*($r + 1$)). Thus, all processes will decide on *flip*($r + 1$) in the first round t such that *flip*(t) $= flip(r + 1)$, and the probability that any process (and thus, a non-faulty one) violates termination is the same as the probability that such a round t never happens, that is, zero. Besides, the expected number of rounds to achieve a round t such that *flip*(t) $= flip(r + 1)$ is 2. Thus, the expected number of rounds of ConsensusS is $3 = E_2(e) = (1/2)(1 + 2) + (1/2)(1 + 2)$.

- $E(e) = E_3(e)$: If all non-faulty processes sent *prefer*(v) and receive no disagreement messages and only *prefer*(v) in round $r+1$, then if *flip*($r+1$) $= v$, all non-faulty processes send *decide*(v) messages and then decide by returning v themselves. Moreover, on receipt of *decide*(v), all remaining processes decide by returning v. If *flip*(r) $\neq v$, then we fall again into the case that all non-faulty processes send *prefer*(v). That is, $E_3(e.2) = E_2(e.2) or E_3(e.2)$. Thus, the probability that any process (and thus, a non-faulty one) violates termination is zero and the expected number of rounds of ConsensusS is $3 = E_3(e) = (1/2)(1 + 1) + (1/2)(1 + 3)$.

In short, in all cases, if $f < n$, the probability that any process (and thus, a non-faulty one) violates termination is zero. Moreover, the expected number of rounds of ConsensusS is 3 for all processes. □

Lemma 5.2. *If $f < n$, each decided value is some process's input.*

Proof. Any decided value v is either an original input or the result of a shared coin flip. Consider the first *prefer*(*flip*(`r`)) statement to be executed, if any. In this case, there must have been a process received both *prefer*(0) and a *prefer*(1) messages, which means that some process had input value 0 and another had input value 1. It follows that either value is some process's input. □

Lemma 5.3. *If $f < n$, no two processes decide differently.*

Proof. Consider the first round r in which a process decides v. It must be the case that at round r, $\mathit{flip}(r) = v$ and all preference messages received by the process are $\mathit{prefer}(v)$. As the messages from all non-faulty processes are received by all processes and there is at least one non-faulty process, all processes receive at least one $\mathit{prefer}(v)$ message, and either decide on v at the same round r or send $\mathit{prefer}(v) = \mathit{prefer}(\mathit{flip}(r))$. It follows that from the next round $r + 1$ on, all messages sent from all processes (and thus, also all received ones) will be $\mathit{prefer}(v)$. Henceforth, no process can decide a value different from v. □

Theorem 5.4. ConsensusS *solves uniform consensus with binary inputs in a synchronous system prone to crashes and send message omissions, with a probability zero of termination violation, and both an optimal constant (3) expected rounds and an optimal $n-1$ resilience (that is, up to $n-1$ processes may be faulty: $f < n$).*

Proof. Follows directly from Lemmas 5.1, 5.2 and 5.3. □

5.6 Optimal Protocol for Send and Receive Omissions

The *ConsensusSR* algorithm in Figure 5.4 solves uniform consensus with binary inputs in optimal 3 expected synchronous rounds tolerating an optimal number of up to $f < n/2$ failures - send message omissions and receive message omissions as well as process crashes.

In ConsensusSR, all processes start by broadcasting their inputs (line 2). Whenever one process does not receive a message from another, it decides that process must be faulty, and ignores it from that point on (line 6). Even so, all non-faulty processes send and receive messages from one another. Moreover, a live faulty process always receives messages from at least one non-faulty process, since otherwise, it would have less than $n/2 + 1$ messages and it would halt before reaching a decision (line 7).

On each round, every process checks if all received messages contain the same preferred value v (line 9). If so, it broadcasts a message that it wants to decide on v (line 10). When receiving this message for the first time (line 12), processes relay it. If a process receives such message from a majority of processes (line 15) or if it receives a message to decide on v (line 18), then it sends messages to all processes to decide on v and retuns v. Note that if a non-faulty process relays the message, all non-faulty processes will relay the message as well, so all non-faulty processes will receive the message from a majority of processes. As every process needs a non-faulty process to relay the message in order to decide on v, if any process decides on v, then every non-faulty process does as well. If a decision is not reached, then the process either sends a message with v as its current preference (line 22), if it received a majority of preferences v, or sends a message containing $\mathit{flip}(r)$ (line 24), otherwise.

Lemma 5.5. *On any single round after initialization (sending the binary private input), only one value is preferred or chosen deterministically.*

Proof. A process prefers or decides v deterministically only if it sees a majority for v. □

Lemma 5.6. *If $f < n/2$, for every process the expected number of rounds of ConsensusSR is 3, and the protocol terminates with probability 1.*

Proof. After initialization (sending the binary private input), if all live processes send $\mathit{prefer}(\mathit{flip}(r))$ or if all live processes send $\mathit{prefer}(v)$, they agree right away, by Lemma 5.5. If some send $\mathit{prefer}(v)$ and some send $\mathit{prefer}(\mathit{flip}(r))$, again by Lemma 5.5, then all live processes will agree in the first round t such that $v = flip(r)$, and the probability that any

```
Recipients = set of all processes;
send prefer(binary preference) to all;
foreach round r {
  Received(r) = set of processes from which messages were received in round r
  Recipients = Recipients intersection Received(r);
  Messages(r) = set of messages received in round r which were sent by Recipients;
  if (|Messages(r)|<n/2+1) {
    halt; // too many failures
    if ( all in Messages(r) are prefer(v)) {
      send want_decide(v) to Recipients;
    }
    on (receipt of want_decide(r,v) for the first time) {
      send want_decide(r,v) to Recipients;
    }
    if (want_decide(r,v) received from majority of processes) {
      send decide(v) to all and return(v);
    }
    on (receipt of decide(v)) {
      send decide(v) to all and return(v);
    }
    if (majority in Messages(r) are prefer(v)) {
      send prefer(v);
    } else {
      send prefer(\flip(r));
    }
}
```

Figure 5.4: Uniform consensus for send and receive message omissions and process crashes.

non-faulty process violates termination is the same as the probability that such a round t never happens, that is, zero. Besides, the expected number of rounds to achieve a round t such that $v = flip(r)$ is 2.

Once agreement by all live processes is achieved, non-faulty processes will receive a majority of $want_decide(r, v)$, send decide(v) and return(v), immediately in the same round. This is because they always receive messages from each other, that is, they always belong to the *Recipients* of non-faulty processes, so once a non-faulty process sends a $want_decide(r, v)$ message, all non-faulty processes will send $want_decide(r, v)$ messages to (and receive them from) all non-faulty processes and guarantee a majority of $want_decide(r, v)$.

In short, in all cases, if $f < n/2$, the probability that a non-faulty process violates termination is zero. Moreover, the expected number of rounds of ConsensusSR is 3 for all processes. □

Lemma 5.7. *If $f < n/2$, all processes in ConsensusSR decide some process's input.*

Proof. A decided value v, from *decide*(v), is just obtained from a *prefer*(v). Now, by induction, a v from *prefer*(v) has to be either an input or a *flip*(r) for some r. However, take the first *prefer*(*flip*(r)) to occur, if any do. In this case, a process received both a *prefer*(0) and a *prefer*(1), which means that there should be a proposed input value equal to 0 and another equal to 1, as the particular *prefer*(*flip*(r)) was the first one to take place. Otherwise, either there would be a majority of *prefer*(v) or Hence, *flip*(r) must be a proposed input value if any *prefer*(*flip*(r)) occurs, and v must also be one of the proposed values. □

Lemma 5.8. *If $f < n/2$, agreement is never violated in ConsensusSR: no two processes decide differently.*

Proof. Consider the first round r when a process decides by returning v. Then, it must be the case that a majority of $want_decide(r, v)$ is received by the process. However, because each process deciding has to receive a $want_decide(r, v)$ from a non-faulty process and non-faulty processes always receive messages from each other, when any process has a majority of $want_decide(r, v)$, it must be the case that all non-faulty processes have a majority of $want_decide(r, v)$, that is, all non-faulty processes decide by returning v as well. □

Theorem 5.9. *ConsensusSR solves uniform consensus with binary inputs in a synchronous system prone to crashes, send message omissions and receive message omissions, with a probability zero of termination violation, and both an optimal constant (3) expected number of rounds and an optimal $n/2 - 1$ resilience (that is, up to $n/2 - 1$ processes may be faulty: $f < n/2$).*

Proof. Follows directly from Lemma 5.6, 5.7 and 5.8. □

Hence, the key idea here is that if secure coprocessors can share secret cryptographic keys (as they do), then they can also share secret seeds for secure pseudo-random number generators. Such shared coins enable randomized (Las Vegas) algorithms for fair exchange and uniform consensus that are optimal in terms of expected running time and resilience.

Chapter 6

Deterministic Message Omission Robust Protocols

Here we present a modular redesign of TrustedPals, a smartcard-based security framework for solving secure multiparty computation (SMC) [36]. TrustedPals is a practical implementation which allows to reduce SMC to the problem of fault-tolerant consensus between smartcards or other trusted coprocessors [17, 54, 53] - an environment where only crashes or message omissions may occur, as in previous chapter. Note that fair exchange is in fact an instance of SMC.

Within the redesign we now continue our exposition of message omission robust protocols by investigating the problem of solving consensus in an asynchronous message omission failure model augmented with failure detectors, instead of the synchronous scenario from previous chapter. To this end, we give novel definitions of both consensus and the class of $\diamond\mathcal{P}$ failure detectors in the omission model and show how to implement $\diamond\mathcal{P}$ and have consensus in such a system with some weak synchrony assumptions. The integration of failure detection into the TrustedPals framework uses tools from privacy enhancing techniques such as message padding and dummy traffic.

The next sections are divided up as follows. In Section 6.1 we give an overview over and motivate the system model of TrustedPals. In Section 6.2 we give details on the system model. In Section 6.3 we define and implement the failure detector $\diamond\mathcal{P}$ in the omission failure model. We then use this failure detector to solve consensus in Section 6.4. In Section 6.5 we describe how to integrate failure detection and consensus securely in the TrustedPals framework.

6.1 Motivation

Consider a set of parties who wish to correctly compute some common function F of their local inputs, while keeping their local data as private as possible, but who do not trust each other, nor the channels by which they communicate. This is the problem of *Secure Multi-party Computation* (SMC) [118]. SMC is a very general security problem, i.e., it can be used to solve various real-life problems such as distributed voting, private bidding and auctions like Ebay, sharing of signature or decryption functions and so on. Unfortunately, solving SMC is—without extra assumptions—very expensive both in terms of communication (number of messages) and time (number of synchronous rounds).

TrustedPals [53] is a smartcard-based implementation of SMC which allows much more efficient solutions to the problem. Conceptually, TrustedPals considers a distributed sys-

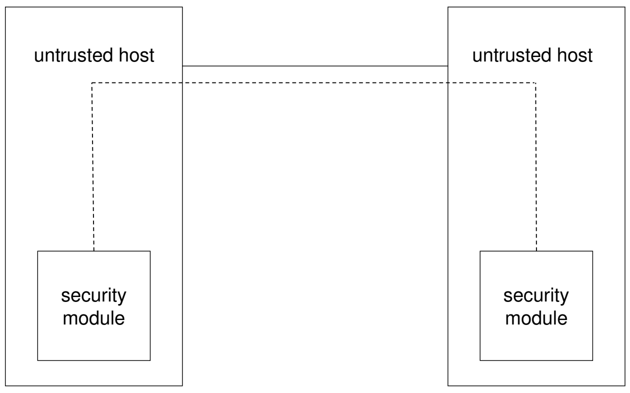

Figure 6.1: Processes with tamper proof security modules.

tem in which processes are locally equipped with tamper proof security modules (see Fig. 6.1) which work as trusted coprocessors, as in previous chapter. In practice, processes are implemented as a Java desktop application and security modules are realized using Java Card Technology enabled smartcards [33]. Roughly speaking, solving SMC between processes is achieved by having the security modules jointly simulate a *trusted third party* (TTP), as we now explain.

To solve SMC in the TrustedPals framework, the function F is coded as a Java function and is distributed within the network in an initial setup phase. Then processes hand their input value to their security module and the framework accomplishes the secure distribution of the input values. Finally, all security modules compute F and return the result to their process. The network of security modules sets up confidential and authenticated channels between each other and operates as a *secure overlay* within the distribution phase. Within this secure overlay, arbitrary and malicious behavior of an attacker is reduced to rather benign faulty behavior (process crashes and message omissions). TrustedPals therefore allows to reduce the security problem of SMC to a fault-tolerant synchronization problem [53], namely that of *consensus.*

To date, TrustedPals assumed the *synchronous* network setting from last chapter, i.e., a setting in which all important timing parameters of the network are known and bounded. This makes TrustedPals sensitive to unforeseen variations in network delay and therefore not very suitable for deployment in networks like the Internet. In this chapter, we explore how to make TrustedPals applicable in environments with less synchrony. More precisely, we unveal the possibilities to implement TrustedPals in a modular fashion inspired by results in fault-tolerant distributed computing: We use an *asynchronous* consensus algorithm and encapsulate (some weak) timing assumptions within a device known as a *failure detector* [28].

The concept of a failure detector has been investigated in quite some detail in systems with merely crash faults [55]. In such systems, correct processes (i.e., processes which do not crash) must eventually permanently suspect crashing processes. There is very little work on failure detection and consensus in message omissions environments. In fact, it is not clear what a sensible definition of a failure detector (and consensus) is in such environments because the notion of a correct process can have several different meanings (e.g., a process with no failures whatsoever or a process which just does not crash but omits messages).

Related Work.

Delporte, Fauconnier and Freiling [38] were the first to investigate non-synchronous settings in the TrustedPals context. Following the approach of Chandra and Toueg [28] (and similarly here) they separate the trusted system into an asynchronous consensus layer and a partially synchronous failure detection layer. They assume that transient omissions are masked by a piggybacking scheme. The main difference however is that they solve a *different version of consensus* than we do: Roughly speaking, message omissions can cause processes to communicate only indirectly, i.e., some processes have to relay messages for other processes. Delporte, Fauconnier and Freiling [38] only guarantee that all processes that can communicate directly with each other solve consensus. In contrast, we allow also those processes which can only communicate indirectly to successfully participate in the consensus. As a minor difference, we focus on the class $\diamond\mathcal{P}$ of eventually perfect failure detectors whereas Delporte, Fauconnier and Freiling [38] implement the less general class Ω. Furthermore, Delporte, Fauconnier and Freiling [38] do not describe how to integrate failure detection within the TrustedPals framework: A realistic adversary who is able to selectively influence the algorithms for failure detection and consensus can cause their consensus algorithm to fail.

Apart from Delporte, Fauconnier and Freiling [38], other authors also investigated solving consensus in systems with omission faults. Unpublished work by Dolev et al. [43] also follows the failure detector approach to solve consensus, however they focus on the class $\diamond\mathcal{S}(om)$ of failure detectors. Babaoglu, Davoli and Montresor [13] also follow the path of $\diamond\mathcal{S}$ to solve consensus in partitionable systems.

Recently, solving SMC *without* security modules has received some attention focusing on two-party protocols [83, 84]. In systems *with* security modules, Avoine and Vaudenay [12] examined the approach of jointly simulating a TTP. This approach was later extended by Avoine et al. [11] who show that in a system with security modules fair exchange can be reduced to a special form of consensus. They derive a solution to fair exchange in a modular way so that the agreement abstraction can be implemented in diverse manners. Note that this solution is deterministic and more costy than optimal randomized ones presented in last chapter [54]. Benenson et al. [17, 53] extended this idea to the general problem of SMC and showed that the use of security modules cannot improve the resilience of SMC but enables more efficient solutions for SMC problems. However, all these works assume a *synchronous* network model.

Correia et al. [35] present a system which employs a real-time distributed security kernel to solve SMC. The architecture is very similar to that of TrustedPals as it also uses the notion of architectural hybridization [111]. However, the adversary model of Correia et al. [35] assumes that the attacker only has remote access to the system while TrustedPals allows the owner of a security module to be the attacker. Like other previous work [12, 17, 11, 54, 53] Correia et al. [35] also assume a synchronous network model at least in a part of the system.

Our work on TrustedPals is closely related to building failure detectors for arbitrary (*byzantine*) failures which has been investigated previously (see for example Kihlstrom, Moser and Melliar-Smith [79] and Doudou, Garbinato and Guerraoui [44]). In contrast to previous work on byzantine failure detectors, we use security modules to avoid the tar pits of this area.

Contributions.

Here we present a modular redesign of TrustedPals using consensus and failure detection as modules. More specifically, we make the following technical contributions:

- We give a novel definition of $\diamond\mathcal{P}$ in the omission model and we show how to implement $\diamond\mathcal{P}$ in a system with weak synchrony assumptions in the spirit of partial synchrony [48].

- We give a new definition of consensus in the omission model and give an algorithm which uses the class $\diamond\mathcal{P}$ to solve consensus. The algorithm is an adaptation of the classic algorithm by Chandra and Toueg [28] for the crash model.

- We integrate failure detection and consensus securely in TrustedPals by employing message padding and dummy traffic, tools known from the area of privacy enhancing techniques.

- We give a detailed security analysis of the system using the attack tree method.

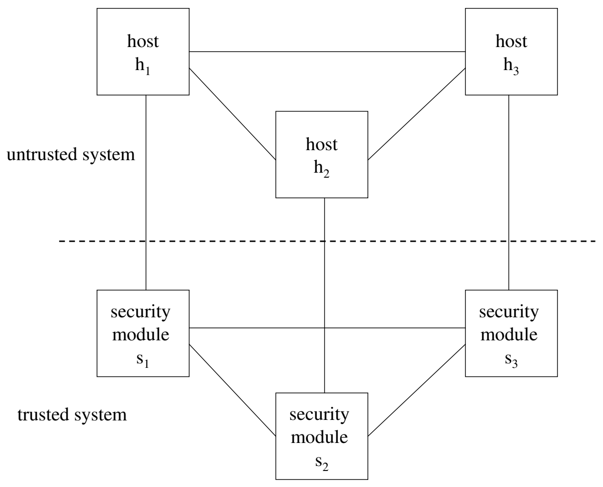

Figure 6.2: The untrusted and trusted system.

6.2 System Model and Architecture

6.2.1 Untrusted and Trusted System

To be able to precisely reason about algorithms and their properties in the TrustedPals system we now formalize the system assumptions within our hybrid model, which is divided into two parts (see Fig. 6.2). Note that, though many aspects from this section may have been already described in previous chapter, there are important details and definitions used here which have not appeared before.

The upper part of our hybrid model consists of n processes which represent the *untrusted hosts*. The lower part equally consists of n processes which represent the security modules. Because of the lack of mutual trust between untrusted hosts, we call the former part the *untrusted system*. Since the security modules trust each other we call the latter part the *trusted system*. Each host is connected to exactly one security module by a direct communication link.

Summarizing, there are two different types of processes: processes in the untrusted system and processes in the trusted system. For brevity, we will use the unqualified term *process* if the type of process is clear from the context.

Within the untrusted system each pair of hosts is connected by a pair of unidirectional communication links, one in each direction. Since the security modules also must use these links to communicate, the trusted system can be considered as an overlay network which is a network that is built on top of another network. Nodes in the overlay network can be thought of as being connected by virtual or logical links. In practice, for example, smartcards could form the overlay network which runs on top of the Internet modeled by the untrusted processes. Within the trusted system we assume the existence of a public key infrastructure, which enables two communicating parties to establish confidentiality, message integrity and user authentication without having to exchange any secret information in advance.

We assume reliable channels, i.e., every message inserted to the channel is eventually delivered at the destination. We assume no particular ordering relation on channels.

6.2.2 Timing Assumptions

We assume that a local clock is available to each host, but clocks are not synchronized within the network. Security modules do not have any clock, they just have a simple step counter, whereby a step consists of receiving a message from other security modules, executing a local computation, and sending a message to other security modules. Passing of time is checked by counting the number of steps executed.

Since trusted and untrusted system operate over the same physical communication channel, we assume the same timing behavior for both systems. Both systems are assumed to be *partially synchronous* meaning that eventually bounds on all important network parameters (processing speed differences, message delivery delay) hold. The model is a variant of the partial synchrony model of Dwork, Lynch and Stockmeyer [48]. The difference is that we assume reliable channels.

We say that a message is *received timely* if it is received after the bounds on the timing parameters hold. Omission of such a message can be reliably detected using timeout-based reasoning.

6.2.3 Failure Assumptions

The model is hybrid because we have distinct failure assumptions for both systems. The failure model we assume in the untrusted system is the *byzantine failure model* [80]. A byzantine process can behave arbitrarily. In the trusted system we assume the failure model of *message omission*, which we now explain again, as previously.

The concept of *omission* faults, meaning that a process drops a message either while sending (*send* omission) or while receiving it (*receive* omission), was introduced by Hadzilacos [61] and later generalized by Perry and Toueg [103]. The failure model used for the trusted system is that of *message omission*, in which processes can crash and experience either send-omissions or receive omissions. We allow the possibility of *transient* omissions, i.e., a process may temporarily drop messages and later on reliably deliver messages again.

A process (untrusted host or security module) is *correct* if it does not fail. A process is *faulty* if it is not correct. We assume a majority of processes to be correct both in the untrusted and in the trusted system. Note that a faulty security module implies a faulty host but a faulty host not necessarily implies a faulty security module.

The motivation behind this hybrid approach is that the system runs in an environment prone to attacks, but the assumptions on the security modules and the possibility to establish secure channels reduce the options of the attacker in the trusted system to attacks on the liveness of the system, i.e., destruction of the security module or interception of messages on the channel.

6.2.4 Classes of Processes in the Trusted System

The omission model in the trusted system implies the possibility of both transient send omissions and receive omissions. Given two processes, p and q, if a single message m sent from p to q is not delivered by q, the following question arises: has p suffered a send omission, or has q suffered a receive omission? Formally, one of the two processes is incorrect, but it is not possible to determine which one. Observe that considering both processes p and q incorrect can be too restrictive. This leads us to reconsider the different classes of

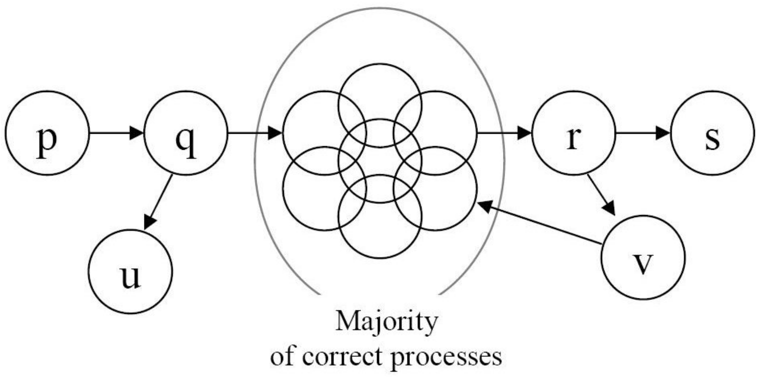

Figure 6.3: Examples for classes of processes.

processes in the omission model with respect to the common correct/incorrect classification. In particular, processes suffering a limited number of omissions, e.g., processes that do not suffer omissions with some correct process, will be considered as *good*, since they can still participate in a distributed protocol like consensus.

On the basis of this motivation, we consider the following two classes of processes:

Definition 6.1. *A process p is* in-connected *if and only if:*

(1) p is a correct process, or

(2) p does not crash and there exists a process q such that q is in-connected and all messages sent by q to p are eventually received timely by p (i.e., q does not suffer any send-omission with p, and p does not suffer any receive-omission with q).

Definition 6.2. *A process p is* out-connected *if and only if:*

(1) p is a correct process, or

(2) p does not crash and there exists a process q such that q is out-connected and all messages sent by p to q are eventually received timely by q (i.e., p does not suffer any send-omission with q, and q does not suffer any receive-omission with p).

Observe that correct processes are both in-connected and out-connected. Observe also that the definitions of in-connected and out-connected processes are recursive. Intuitively, there is a timely path with no omissions from every correct process to every in-connected process. Also, there is a timely path with no omissions from every out-connected process to every correct process, and hence to every in-connected process.

Fig. 6.3 shows an example. In the figure, arcs represent timely links with no omissions (they are not shown for the majority of correct processes). Processes p and q are out-connected, while process s is in-connected, and processes r and v are both in-connected and out-connected. Finally, process u is neither in-connected nor out-connected.

6.2.5 The TrustedPals Architecture

Fig. 6.4 shows the layers and interfaces of the proposed modular architecture for TrustedPals. A message exchange is performed on the transport layer, which is under control of

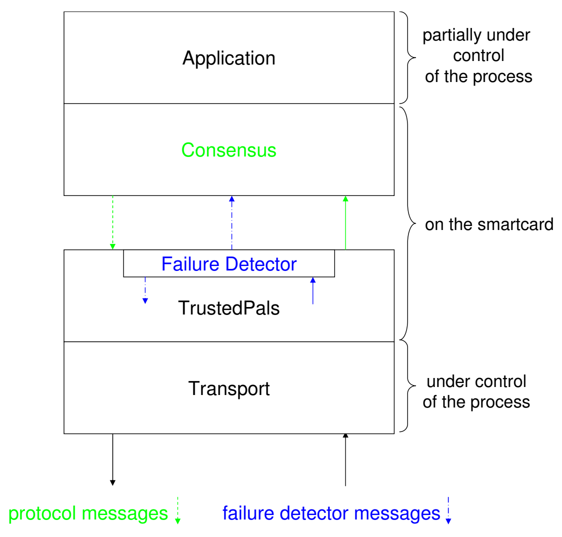
Application
partially under control of the process
Consensus
on the smartcard
Failure Detector
TrustedPals
Transport
under control of the process
protocol messages
failure detector messages

Figure 6.4: The architecture of our system.

the untrusted host. The failure detector and the security mechanisms for message encryption etc. run in the TrustedPals layer. In the consensus layer runs the consensus algorithm. On the application layer, which again is under the control of the untrusted host, protocols like fair exchange operate.

6.3 Failure Detection in TrustedPals

Based on the two new classes of processes defined in the previous section, we redefine now the properties that $\diamond\mathcal{P}$ must satisfy in the omission model. While the common correct/faulty classification of processes is well addressed by means of a list of suspected processes, in the omission model we will consider two lists of processes, one for the in-connected processes and the other one for the out-connected processes. If a process p has a process q in its list of in-connected (out-connected) processes, we say that p *considers* q *as in-connected* (*out-connected*). The $\diamond\mathcal{P}$ class of failure detectors in the omission model satisfies the following properties:

- *Strong Completeness.* Eventually every process that is not out-connected will be permanently considered as not out-connected by every in-connected process.
- *Eventual Strong Accuracy.* Eventually every process that is out-connected will be permanently considered as out-connected by every in-connected process.
- *In-connectivity.* Eventually every process that is in-connected will permanently consider itself as in-connected.

Figs. 6.5, 6.6 and 6.7 present an algorithm implementing $\diamond\mathcal{P}$. The algorithm provides to every process p a list of in-connected processes, $InConnected_p$, and another list of out-connected processes, $OutConnected_p$. For every in-connected process p, these lists will have the information required to satisfy the properties of $\diamond\mathcal{P}$. In particular, the list $OutConnected_p$ will eventually and permanently contain exactly all the out-connected processes. Regarding the $InConnected_p$ list, it will eventually and permanently contain p itself. as well as every process that is both in-connected and out-connected (hence, at least all correct processes).

In order to detect message omissions, messages carry a sequence number. Besides, every process p uses a matrix M_p of $n \times n$ elements. In the beginning, all processes are supposed to be correct, so every element in the matrix has a value of 1. If all messages sent from a process q to a process p are received timely by p, $M_p[p][q]$ will be maintained to 1. Otherwise, process p will set $M_p[p][q]$ to 0. In this way, the matrix will have the information needed to calculate the lists of in-connected and out-connected processes.

Actually, M represents the transposed adjacency matrix of a directed graph, where the value of the element $M[p][q]$ shows if there is an arc from q to p. We can derive from powers of the adjacency matrix if there is a path with no omission of any length between every pair of processes. Observe that in the given algorithm a process does not monitor itself and, as a consequence, the elements of the main diagonal of the matrix are always set to 1. Taking this into account, the n-th power of the adjacency matrix, $A_p = (M_p)^n$, gives us the information we need to obtain the sets of in-connected and out-connected processes. A process p is in-connected if it is able to receive all the messages (either directly or indirectly) from at least $\lceil \frac{(n+1)}{2} \rceil$ processes. Similarly, a process p is out-connected if at least $\lceil \frac{(n+1)}{2} \rceil$ processes are able to receive (either directly or indirectly) all the messages sent by p. The lists of in-connected and out-connected processes are computed in the *update_In_Out_Connected_lists*() procedure, which is called every time a value of the matrix M_p is changed.

In Task 1 (line 14), a process p periodically sends a heartbeat message to the rest of processes. When a message is sent, the sequence number associated to the destination is incremented. Observe that the matrix M_p is sent in the heartbeat messages.

In Task 2 (line 21), if a process p does not receive the next expected message from a process q in the expected time, the value of $M_p[p][q]$ is set to 0.

In Task 3 (line 28), received messages are processed. The messages a process p receives from another process q are delivered following the sequence number $next\_receive_p[q]$. Every process p has a buffer for every other process q to store unordered messages received from q. If p receives a message from q with a sequence number different from the expected one, this message is inserted in $Buffer_p[q]$ and the message is not delivered yet (line 42). A message is delivered when it is the next expected message, either because it has been just received (line 30) or it is inside the buffer (line 33). If the delivered message was in the buffer, it is removed from there. Having delivered the next expected message from a process q, if the buffer is empty it means that there is no message left from q, so $M_p[p][q]$ is set to 1. This way, process p fills its corresponding row in the matrix indicating if all the messages it expected from every other process have been received timely.

The procedure *deliver_next_message*() is used to update the adjacency matrix M_p using the information carried by the message. In the procedure, process p copies into M_p the row q of the matrix M_q received from q. This way, p learns about q's input connectivity. With respect to every other process u, a mechanism based on version numbers is used to avoid copying old information about u's input connectivity. Process p will only copy into M_p the row u of M_q if its version number is higher.

Correctness Proof

Lemma 6.3. *$\forall p, q \in \Pi$, if neither p nor q crashes and there is no message omission from q to p, eventually and permanently $M_p[p][q]$=1. Otherwise, i.e., if q crashes or there is*

```
Procedure main()
    $InConnected_p \leftarrow \Pi$
    $OutConnected_p \leftarrow \Pi$
    forall $q \in \Pi - \{p\}$ do
        $\Delta_p(q) \leftarrow$ default time-out interval   {$\Delta_p(q)$ denotes the duration of $p$'s time-out interval for $q$}
        $next\_send_p[q] \leftarrow 1$   {sequence number of the next message sent to $q$}
        $next\_receive_p[q] \leftarrow 1$   {sequence number of the next message expected from $q$}
        $Buffer_p[q] \leftarrow \emptyset$

    forall $q \in \Pi$ do
        forall $u \in \Pi$ do
            $M_p[q][u] \leftarrow 1$   {$M_p[q][u] = 0$ means that $q$ has not received at least one message from $u$}
        $Version_p[q] \leftarrow 0$   {$Version_p$ contains the version number for every row of $M_p$}

    $UpdateVersion \leftarrow$ false
     || Task 1: repeat periodically
        if $UpdateVersion$ then   {$p$'s row has changed}
            $Version_p[p] \leftarrow Version_p[p] + 1$
            $UpdateVersion \leftarrow$ false

        forall $q \in \Pi - \{p\}$ do
            send $(ALIVE, p, next\_send_p[q], M_p, Version_p)$ to $q$   {sends a heartbeat}
            $next\_send_p[q] \leftarrow next\_send_p[q] + 1$   {$p$ updates its sequence number for $q$}

     || Task 2: repeat periodically
        if ( $p$ did not receive $(ALIVE, q, next\_receive_p[q], M_q, Version_q)$
             from $q \neq p$ during the last $\Delta_p(q)$ ticks of $p$'s clock ) then
        {the next message in the sequence has not been received timely}
            $\Delta_p(q) \leftarrow \Delta_p(q) + 1$
            if $M_p[p][q] = 1$ then
                $M_p[p][q] \leftarrow 0$   {the potential omission is reflected in $M_p$}
                $UpdateVersion \leftarrow$ true
                call update_In_Out_Connected_lists()

     || Task 3: when receive $(ALIVE, q, c, M_q, Version_q)$ for some $q$
        if $c = next\_receive_p[q]$ then   {it is the next message expected from $q$}
            call deliver_next_message($q$, $M_q$, $Version_q$)   {the message is delivered}
            $next\_receive_p[q] \leftarrow next\_receive_p[q] + 1$
            while $(ALIVE, q, next\_receive_p[q], M_q, Version_q) \in Buffer_p[q]$ do
                call deliver_next_message($q$, $M_q$, $Version_q$)
                remove $(ALIVE, q, M_q, next\_receive_p[q], Version_q)$ from $Buffer_p[q]$
                $next\_receive_p[q] \leftarrow next\_receive_p[q] + 1$

            if $Buffer_p[q] = \emptyset$ then
                $M_p[p][q] \leftarrow 1$   {so far $p$ has received all messages from $q$}
                $UpdateVersion \leftarrow$ true

            if $M_p$ has changed then
                call update_In_Out_Connected_lists()

        else
            insert $(ALIVE, q, c, M_q, Version_q)$ into $Buffer_p[q]$
```

Figure 6.5: $\Diamond\mathcal{P}$ in the omission model: main algorithm.

```
Result: $InConnected_p$ and $OutConnected_p$ lists
Procedure update_In_Out_Connected_lists()
    $A_p \leftarrow (M_p)^n$                                   {$A_p$ is the $n$-th power of the $M_p$ matrix}
    forall $u, v \in \Pi$ do
        if $A_p[u][v] > 0$ then
            $A_p[u][v] \leftarrow 1$
    $In \leftarrow \emptyset$
    $Out \leftarrow \emptyset$
    forall $q \in \Pi$ do
        if $(\sum_{i=0}^{n-1} A_p[q][i] \geq \lceil \frac{(n+1)}{2} \rceil)$ then
            $In \leftarrow In \cup \{q\}$
        if $(\sum_{i=0}^{n-1} A_p[i][q] \geq \lceil \frac{(n+1)}{2} \rceil)$ then
            $Out \leftarrow Out \cup \{q\}$
    $InConnected_p \leftarrow In$
    $OutConnected_p \leftarrow Out$
```

Figure 6.6: $\diamond\mathcal{P}$ in the omission model: procedure update_In_Out_Connected_lists().

```
Input: $q$: process from which the message has been received; $M_q$: $q$'s knowledge about the
       system; $Version_q$: version number of each row of $M_q$
Result: update of $M_p$ matrix and $Version_p$ vector
Procedure deliver_next_message()
    forall $v \in \Pi$ do                               {$q$'s row of $M_q$ is systematically copied into $M_p$}
        $M_p[q][v] \leftarrow M_q[q][v]$
    forall $u \in \Pi - \{p, q\}$ do
        if $Version_q[u] > Version_p[u]$ then            {$q$'s information about $u$ is more recent than
        $p$'s}
            forall $v \in \Pi$ do
                $M_p[u][v] \leftarrow M_q[u][v]$
            $Version_p[u] \leftarrow Version_q[u]$
```

Figure 6.7: $\diamond\mathcal{P}$ in the omission model: procedure deliver_next_message().

some message omission from q to p, eventually and permanently $M_p[p][q]$=0.

Proof. If all messages sent by q to p are eventually received by p, every heartbeat message sent by Task 1 of q will be eventually received by Task 3 of p. Observe that the *deliver_next_message*() procedure does not modify $M_p[p][q]$. By Task 2 of p, $M_p[p][q]$ is set to 0 when the next expected message m from q has not been received timely by p. Eventually, on the reception of m by Task 3 of p, m will be delivered. After that, if $Buffer_p[q]$ becomes empty, $M_p[p][q]$ will be set to 1. Observe that this will permanently happen when $\Delta_p(q)$ reaches the unknown bound on message transmission time. Since $\Delta_p(q)$ is incremented when m is not received timely, and since the communication link between q and p is eventually timely, this bound will be eventually reached, after which p will receive every $(ALIVE, q, -, -, -)$ message always before $\Delta_p(q)$ expires, and $M_p[p][q] = 1$ forever.

On the other hand, if a message m is omitted from q to p, by Task 2 of p, $M_p[p][q]$ is set to 0 because m has not been received timely by p. After that, p could receive a message l with a higher sequence number from q. This message would be inserted in $Buffer_p[q]$ because it was not the next expected message. If the next expected message m is never received, $Buffer_p[q]$ will never become empty and $M_p[p][q] = 0$ forever. Observe that if p does not receive more messages from q after the omission of m because all the messages are omitted or q has crashed, Task 3 of p will never be executed due to a reception of a message from q, and $M_p[p][q] = 0$ forever too. □

Lemma 6.4. $\forall p, q \in \Pi$, *if neither p nor q crashes and there is a path with no omission from q to p, eventually and permanently the row q of matrices M_p and M_q will be identical,* *i.e.*, $M_p[q][] = M_q[q][]$.

Proof. Process q has information about its own in-connectivity in the row q of its matrix $M_q[q][]$. Every time a value of this row changes, its version number will be incremented. By Lemma 6.3, this information eventually stabilizes in q and its version number will be the highest associated to this row in the system. When a process r receives a message from q it will copy the row $M_q[q][]$ into $M_r[q][]$ if the row version number is higher, that is to say, the received information is newer. If there is no message omission from process q to process r, r will obtain the last version of the in-connectivity information of q. After updating the information about q, by Task 1 r will send this information in its own matrix to the rest of processes.

If there is a path with no omission from q to p, recursively, p will receive the latest information about q going through all the processes in the path. The version number mechanism ensures that old information about q arriving at p will not be copied into M_p. As a consequence, $M_p[q][] = M_q[q][]$ eventually and permanently. □

Lemma 6.5. *$\forall p, q \in \Pi$, if neither p nor q crashes and there is a path with no omission from q to p, eventually and permanently $(M_p)^n[p][q] \geq 1$.*

Proof. The path from q to p will be composed of processes $q, u, v, ..., p$. If this is a path with no omission, there is no omission from q to u, from u to v and so on. By Lemma 6.3, $M_u[u][q] = 1$, $M_v[v][u] = 1$ and so on. By Lemma 6.4, the rows $q, u, v, ..., p$ of the adjacency matrix M_p will be eventually updated with the in-connectivity information of all these processes, reflecting this path in the matrix. This way, the n-th power of the adjacency matrix will tell us that there is a path of some length from q to p, being $(M_p)^n[p][q] \geq 1$. □

Lemma 6.6. *$\forall p$ in-connected, $\forall q$ out-connected it holds that eventually and permanently $q \in OutConnected_p$.*

Proof. By definition, since process p is in-connected there is a path with no omission from some correct process to p. By definition too, since process q is out-connected there is a path with no omission from q to some correct process. Observe that all correct processes never suffer any omission, and considering that a majority of processes is correct, by Lemma 6.3, every correct process will have at least $\lceil \frac{(n+1)}{2} \rceil$ 1 values in its row. Even more, by Lemma 6.4, every correct process will update its matrix copying all the rows of the rest of correct processes, having eventually and permanently at least $\lceil \frac{(n+1)}{2} \rceil$ 1 values in its column. Considering that there is no omission among correct processes, we can derive that there is a path with no omission from q to every correct process and also that there is a path with no omission from every correct process to p. By Lemma 6.5, for every correct process r, $(M_r)^n[r][q] \geq 1$. Being a path with no omission from all correct processes to p, by Lemma 6.4, p will update its matrix from the correct processes, and $(M_p)^n[][q] \geq 1$ for more than $\lceil \frac{(n+1)}{2} \rceil$ processes (at least the correct processes). As a consequence, according to the procedure *update_In_Out_Connected_lists*(), q will be permanently included in the list $OutConnected_p$. □

Lemma 6.7. *$\forall p$ in-connected, $\forall q$ not out-connected, eventually and permanently $q \notin OutConnected_p$.*

Proof. Since q is not out-connected, it does not exist a correct process r such that there is a path with no omission from q to r. By Lemma 6.4, if p is in-connected, p will eventually and permanently know about the connectivity of every correct process r, and therefore,

$(M_p)^n[r][q] = 0$. Since there is a majority of correct processes, the number of processes such that $(M_p)^n[][q] \geq 1$ will always be less than $\lceil \frac{(n+1)}{2} \rceil$, so eventually and permanently p will consider q as not out-connected ($q \notin OutConnected_p$). ☐

Lemma 6.8. *$\forall p$ in-connected, eventually and permanently $p \in InConnected_p$.*

Proof. As shown in Lemma 6.6, there is a a path with no omission from every correct process to p. Considering that there is a majority of correct processes, the number of processes with a path with no omission to p will be at least $\lceil \frac{(n+1)}{2} \rceil$, and by Lemma 6.5, p will eventually and permanently consider itself as in-connected in the procedure update_In_Out_Connected_lists(). ☐

Theorem 6.9. *The algorithm of Figure 6.5 implements $\diamond\mathcal{P}$ in the omission model.*

Proof. The strong completeness, eventual strong accuracy, and in-connectivity properties of $\diamond\mathcal{P}$ are satisfied by Lemmas 6.7, 6.6, and 6.8 respectively. ☐

6.4 $\diamond\mathcal{P}$-based Consensus in TrustedPals

In the *consensus* problem, every process proposes a value, and correct processes must eventually decide on some common value that has been proposed. In the crash model, every *correct process* is required to eventually decide some value. This is called the *Termination* property of consensus. In order to adapt consensus to the omission model, we argue that only the Termination property has to be redefined. This property involves now every in-connected process, since, despite they can suffer some omissions, in-connected processes are those that will be able to decide.

The properties of consensus in the omission model are the following:

- *Termination.* Every *in-connected* process eventually decides some value.
- *Integrity.* Every process decides at most once.
- *Uniform agreement.* No two processes decide differently.
- *Validity.* If a process decides v, then v was proposed by some process.

Figs. 6.8 and 6.9 present an algorithm solving consensus using $\diamond\mathcal{P}$ in the omission model. It is an adaptation of the well-known Chandra-Toueg consensus algorithm. Instead of explaining the algorithm from scratch, we just comment on the modifications required to adapt the original algorithm:

- In Phase 2, the current coordinator waits for a majority of estimates while it considers itself as in-connected in order not to block. Only in case it receives a majority of estimates a valid estimate is sent to all. If it is not the case, the coordinator sends a *NEXT* message indicating that the current round cannot be successful.

- In Phase 3, every process p waits for the new estimate proposed by the current coordinator while p considers itself as in-connected and the coordinator as out-connected in order not to block. Also, p can receive a *NEXT* message indicating that the current round cannot be successful. In case p receives a valid estimate, it replies with a *ack* message. Otherwise, p sends a *nack* message to the current coordinator.

```
     {Every process p executes the following}
(1)  Procedure propose($v_p$)
(2)      $estimate_p \leftarrow v_p$                 {$estimate_p$ is p's estimate of the decision va
(3)      $state_p \leftarrow undecided$
(4)      $r_p \leftarrow 0$                                  {$r_p$ is p's current round num
(5)      $ts_p \leftarrow 0$            {$ts_p$ is the last round in which p updated $estimate_p$, initial

         {Rotate through coordinators until decision is reached}
(6)      while $state_p = undecided$ do
(7)          $r_p \leftarrow r_p + 1$
(8)          $c_p \leftarrow (r_p \bmod n) + 1$                  {$c_p$ is the current coordina
(9)          Phase 1: {All processes p send $estimate_p$ to the current coordinator}
(10)             send $(p, r_p, estimate_p, ts_p)$ to $c_p$

(11)         Phase 2:
             { The current coordinator tries to gather $\lceil \frac{(n+1)}{2} \rceil$ estimates. If it succeeds,
               it proposes a new estimate. Otherwise, it sends a $NEXT$ message to all }
(12)             if $p = c_p$ then
(13)                 wait until
                     ( ($p \in \Pi - InConnected_p$) or
                       (for $\lceil \frac{(n+1)}{2} \rceil$ processes $q$: received $(q, r_p, estimate_q, ts_q)$ from $q$) )
(14)                 if for $\lceil \frac{(n+1)}{2} \rceil$ processes $q$: received $(q, r_p, estimate_q, ts_q)$ from $q$ th
(15)                     $success_p \leftarrow TRUE$
(16)                     $msgs_p[r_p] \leftarrow \{(q, r_p, estimate_q, ts_q) \mid p$ received
                         $(q, r_p, estimate_q, ts_q)$ from $q\}$
(17)                     $t \leftarrow$ largest $ts_q$ such that $(q, r_p, estimate_q, ts_q) \in msgs_p[r_p]$
(18)                     $estimate_p \leftarrow$ select one $estimate_q$ such that $(q, r_p, estimate_q, t$
                         $\in msgs_p[r_p]$
(19)                     send $(p, r_p, estimate_p)$ to all
(20)                 else
(21)                     $success_p \leftarrow FALSE$
(22)                     send $(p, r_p, NEXT)$ to all

(23)         Phase 3: {All processes wait for the new estimate proposed by the coordinator}
(24)             wait until
                 ( ($p \in \Pi - InConnected_p$) or
                   received [ $(c_p, r_p, estimate_{c_p})$ or $(c_p, r_p, NEXT)$ ] from $c_p$ or
                   ($c_p \in \Pi - OutConnected_p$) )
(25)             if received $(c_p, r_p, estimate_{c_p})$ from $c_p$ then
(26)                 $estimate_p \leftarrow estimate_{c_p}$
(27)                 $ts_p \leftarrow r_p$
(28)                 send $(p, r_p, ack)$ to $c_p$
(29)             else
(30)                 send $(p, r_p, nack)$ to $c_p$

(31)         Phase 4:
             { If the current coordinator sent a valid estimate in Phase 2, it waits for replies o
               out-connected processes while it considers itself as in-connected. If $\lceil \frac{(n+1)}{2} \rceil$
               processes replied with $ack$, the coordinator R-broadcasts a decide message
(32)             if ($p = c_p$) and ($success_p = TRUE$) then
                                  [ ($p \in \Pi - InConnected_p$) or                          ]
                     wait until   [                      ( received $(q, r_p, ack)$ or    )   ]
                                  [ for all process $q$: ( received $(q, r_p, nack)$ or   )   ]
(33)                              [                      ( $q \in \Pi - OutConnected_p$   )   ]
(34)                 if for $\lceil \frac{(n+1)}{2} \rceil$ processes $q$:received $(q, r_p, ack)$ then
(35)                     R-broadcast($p, r_p, estimate_p, decide$)
```

Figure 6.8: Solving consensus in the omission model using $\Diamond\mathcal{P}$: main algorithm.

```
     {If p R-delivers a decide message, p decides accordingly}
(36) when R-deliver($q, r_q, estimate_q, decide$) do
(37)     if $state_p = undecided$ then
(38)         $decide(estimate_q)$
(39)         $state_p \leftarrow decided$
```

Figure 6.9: Solving consensus in the omission model using $\Diamond\mathcal{P}$: adopting the decision.

- In Phase 4, if the current coordinator sent a valid estimate in Phase 2, it waits for replies of out-connected processes while it considers itself as in-connected in order not to block. If a majority of processes replied with *ack*, the coordinator R-broadcasts a decide message.

When a process p sends a consensus message m to another process q, the following approach is assumed: (1) p sends m to all processes, including q, except p itself, and (2) whenever p receives for the first time a message m whose destination is another process q different from p, p forwards m to all processes (except the process from which p has received m and p itself). Clearly, this approach can take advantage of the underlying all-to-all implementation of the $\Diamond\mathcal{P}$ failure detector.

Correctness Proof

We provide here a proof sketch of our adapted consensus algorithm in the omission model. First of all, observe that uniform agreement is preserved, because we keep the original mechanism based on majorities to decide on a value. Also, it is easy to see that integrity and validity are satisfied. Finally, in order to show that termination is satisfied, we first show that the algorithm does not block in any of its **wait** instructions:

- In Phase 2, if the current coordinator p is not in-connected, it will eventually stop waiting because the failure detector will eventually exclude p from $InConnected_p$. On the other hand, if p is in-connected, it will eventually receive a majority of estimates since there is a majority of correct processes in the system. Hence, no coordinator blocks forever in the **wait** instruction of Phase 2.
- In Phase 3, every process p waits for the new estimate proposed by the current coordinator or a *NEXT* message while p considers itself as in-connected and the coordinator as out-connected. Clearly, by the properties of $\Diamond\mathcal{P}$ no process blocks forever in the **wait** instruction of Phase 3.
- In Phase 4, the current coordinator waits for replies of out-connected processes while it considers itself as in-connected. Again, by the properties of $\Diamond\mathcal{P}$ and the fact that there is a majority of correct processes in the system, no coordinator blocks forever in the **wait** instruction of Phase 4.

By the previous, eventually some correct process c will coordinate a round in which:

- In Phase 2, c will receive a majority of estimates, because c will be permanently in $InConnected_c$ (by the properties of $\Diamond\mathcal{P}$) and there is a majority of correct processes in the system. Hence, c will send a valid estimate to all processes at the end of Phase 2.
- In Phase 3, every correct process p will receive c's valid estimate, because p will be permanently in $InConnected_p$ and c will be permanently in $OutConnected_p$ (by the properties of $\Diamond\mathcal{P}$). Hence, p will send a *ack* message to c at the end of Phase 3.
- In Phase 4, c will receive a majority of *ack* messages, because c will be permanently in $InConnected_c$ and all correct processes will be permanently in $OutConnected_c$ (by the properties of $\Diamond\mathcal{P}$) and there is a majority of correct processes in the system. Hence, c will R-broadcast the decision, and every in-connected process will eventually decide.

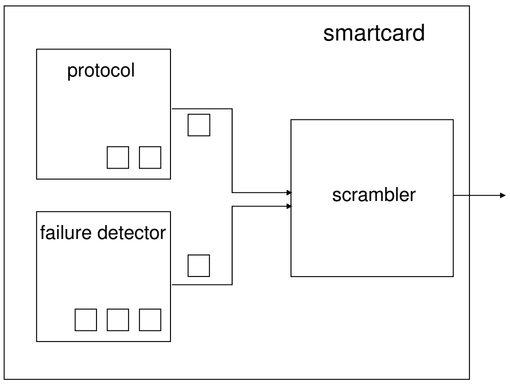

Figure 6.10: Smartcard with scrambler.

6.5 Integrating Failure Detection and Consensus Securely

As depicted in Fig. 6.4, the TrustedPals layer receives messages from the consensus protocol and from the failure detector. If an untrusted host could distinguish protocol messages from failure detector messages he could intercept all former messages while leaving the latter untouched. This would result in a failure detector working properly but a consensus protocol to block forever. In order to prevent such malicious actions we piggyback the protocol messages on the failure detector messages, which are sent in regular time intervals. To make sure that the adversary can not distinguish the packets with the protocol message piggybacked from the ones without protocol message, packets will have the same size, i.e., failure detector messages are padded and protocol messages are divided into a predefined length. It might be inefficient for small messages to be padded or large packets split up in order to get a message of the desired size. However, it is necessary to find an acceptable tradeoff between security and performance such that a message size provides better security in expense of worse performance.

We assume a *scrambler* which receives the protocol and failure detector messages and outputs equal looking messages of the same size in regular time intervals (see Fig. 6.10). It proceeds as follows. Whenever a protocol message has to be sent, it will be piggybacked on the failure detector message. If there is no protocol message ready to be sent, the packet's payload will be filled with random bits. In order to be efficient, the predefined size of the messages sent will be kept as small as possible. If a protocol message is too big, it will be divided, using a fragmentation mechanism, and piggybacked into multiple failure detector messages. Since the protocol is asynchronous, even long delays can be tolerated as long as the failure detector works correctly.

Cryptography is applied to prevent and detect cheating and other malicious activities. We use a public key cryptosystem for encryption. Each message m in our model will be signed and then encrypted in order to reach authenticity, confidentiality, integrity, and non-repudiation.

The source and destination address are encrypted because this enables the receiver

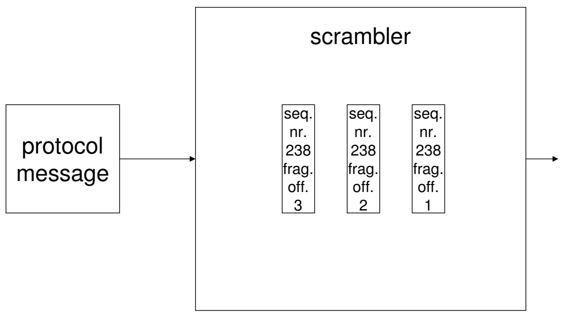
scrambler
protocol message
seq. nr. 238 frag. off. 3
seq. nr. 238 frag. off. 2
seq. nr. 238 frag. off. 1

Figure 6.11: Example of scrambler's function.

of a message to check whether the received message was intended for it or not and who the sender was. Thus, a malicious process cannot change the destination address in the header of a message from its security module and send it to an arbitrary destination without being detected. To detect a message deletion or loss, each message which is sent gets an identification number, where the fragment offset field determines the place of a particular fragment in the original message with same identification number.

As an example for the scrambler's function, consider the situation where the scrambler takes a protocol message m, whose size is three times the size of a failure detector message, from the queue of protocol messages to be sent. The scrambler divides the protocol message in three parts and assigns the next available sequence number to each part. Also each part gets a fragment offset. The first message part gets the fragment offset 1, the second message part gets the fragment offset 2, and the last message part gets the fragment offset 3. (see Fig. 6.11). Next, the sequence number, all other fields, and the first message part all together are signed with the private key of the sender. After that, the signature is encrypted. Then, the next failure detector message is taken from the queue of failure detector messages to be sent and the encrypted message part is inserted into the failure detector message payload. Now, the first message part is ready to be sent in the next upcoming interval. The same is applied to the second and third part of the protocol message.

When the queue of protocol messages is empty, the scrambler only takes a failure detector message from the queue of failure detector messages (see Fig. 6.12) . Here, also a sequence number is assigned. But the fragment offset and the data field are filled with random bits. Then, the sequence number, the not set CD field, the not set MF field, the source and destination address of the message, and the random bits are taken and signed all together. Now, the signature is encrypted and added to the failure detector message payload. Then, the message can be sent in the next upcoming interval.

```
if (queue of protocol messages is empty) {
    take a failure detector message
    assign a sequence number
    fill fragment offset field and data field with random bits
    sign message payload
    encrypt signature
    add ciphertext into failure detector message payload
    send generated message in next upcoming interval
}
```

Figure 6.12: Part of scrambler's function in pseudocode.

6.6 Security Evaluation

The correct execution and termination of the algorithm must be provided and all parties must have the confidence that certain objectives associated with the algorithm's security have been met. The system must provide reliable multi-party interaction under partial synchrony and subject to malicious as well as accidental faults. We evaluate if all desired security objectives are accomplished in the proposed system model. Since the failure model assumed in the untrusted system is the Byzantine failure model, malicious processes can collude, exchange information, and jointly gather knowledge to perform any kind of attack on the system. On the other hand, correct processes try to achieve safety and liveness and keep the messages exchanged secret.

To model the security threats against the system we make use of attack trees. Attack trees are conceptual diagrams of threats on systems and possible attacks to reach those threats. The reason we have chosen attack trees and not formal proofs to perform a security analysis is because verifying information flow properties is complex and different from proving safety and liveness properties

In the next section, we introduce three attack trees. The leafs of the trees are used as a basis for further discussion, where the attacks they represent are analyzed in more detail, including capabilities needed by the attacker.

6.6.1 Analysis

In this section we perform an analysis of the proposed system using the methodology of attack trees. We have to examine the following attacker goals:

Goal 1: Violate safety properties of the system

Goal 2: Violate liveness properties of the system

Goal 3: Violate information flow properties of the system

In the following we will give attack trees for each of these goals.

Attacks Aimed at Safety Properties of the System

Figure 6.13 shows an attack tree for the threat of the system's safety properties. The goal is to violate the safety properties of the system. In order to violate the safety properties of the system, the attacker can cause the safety properties of the consensus algorithm to fail. This can be done either by violating the validity property of the algorithm or by violating the agreement property of the algorithm, or attacking the failure detector. The validity property can be violated by attacking the integrity of the system. The agreement property can also be violated by attacking the integrity of the system or by deleting and

Goal 1: Violate safety properties of the system

1. Cause safety properties of consensus to fail
    - (a) Violate validity property (OR)
        - i. Attack integrity
            - A. Manipulate message payload (OR)
                - 1. Cryptoanalyze asymmetric encryption (OR)
                - 2. Flip some bits (OR)
                - 3. Replace some blocks with previously sent message blocks
            - B. Manipulating message header
                - 1. Spoofing
    - (b) Violate agreement property (OR)
        - i. Attack integrity (OR)
        - ii. Delete messages (AND)
        - iii. Inject false messages (OR)
        - iv. Inject replays of previous messages
    - (c) Attack failure detector
        - i. Violate eventual strong accuracy property
            - A. Delete messages $*$ (OR)
            - B. Attack processes' availability $*$

Figure 6.13: Attack tree for the threat of safety properties.

injecting messages. To attack the failure detector, the attacker has to violate the eventual strong accuracy property of the failure detector. Note that it can be detected if messages were corrupted in the network and corruptions are converted to omission failures.

Attacks Aimed at Liveness Properties of the System

Figure 6.14 shows an attack tree for the threat of the system's liveness properties. The goal is to violate the liveness properties of the system. For this purpose, an attacker can violate the liveness properties of the consensus algorithm. In order to attack the liveness properties of the consensus algorithm, it either has to violate the termination property of the algorithm or violate the failure detector's strong completeness property. To violate the termination property of consensus the attacker must try to avoid that the majority of correct processes eventually decide on some value by attacking the availability of either the network or the processes or the smartcard.

Attacks Aimed at Information Flow Properties of the System

Figure 6.15 shows an attack tree for the threat of the system's information flow properties. The goal is to violate the information flow properties of the system. For this purpose, attackers can attack the system's cofidentiality or do traffic flow analysis. To attack confidentiality, they either can try to read the encrypted messages transmitted over the network or can attack their own smartcard. There are many ways to read encrypted messages. Here, we consider only some of the most common ones. Note that in the

Goal 2: Violate liveness properties of the system

1. Cause liveness properties of consensus to fail
    - (a) Violate termination property (OR)
        - i. Attack availability
            - A. Attack network (OR)
                1. Physical destruction of network (OR)
                2. Denial of service attack on network
            - B. Attack process (OR)
                1. Physical destruction of processes (OR)
                2. Denial of service attack on processes
            - C. Attack smartcard
                1. Physical destruction of the smartcard (OR)
                2. Denial of service attack on the smartcard (OR)
                3. Pull smartcard out from smartcard reader
    - (b) Attack failure detector
        - i. Violate strong completeness property
            - A. Inject false messages (OR)
            - B. Inject replays of previous messages

Figure 6.14: Attack tree for the threat of liveness properties.

majority of cryptographic systems, the secrecy of the method to encrypt data is based on the encryption algorithm, which is the collection of the mathematical rules determining the sequence of fulfilling the elementary operations above the data, and on the cryptographic key, which determines the precise computation of the plaintext in ciphertext and vice versa.

Goal 3: Violate information flow properties of the system

1. Attack Confidentiality (OR)
    (a) Read encrypted message in transfer (OR)
        i. Decrypt the message itself (OR)
            A. Mathematically break asymmetric encryption (OR)
            B. Ciphertext-only attack
        ii. Obtain private key of recipient (OR)
            A. Brute-force attack (OR)
            B. Mathematically break asymmetric encryption (OR)
            C. Social engineering (OR)
            D. Ciphertext-only attack (OR)
            E. Known-plaintext attack
        iii. Get recipient to (help) decrypt the message
            A. Chosen-plaintext attack (OR)
            B. Adaptive chosen-plaintext attack (OR)
            C. Chosen-ciphertext attack (OR)
            D. Adaptive chosen-ciphertext attack (OR)
            E. Read message after it is decrypted by the recipient
    (b) Attack smartcard
        i. Side-channel attack (OR)
        ii. Physical attack
2. Traffic flow analysis
    (a) Analyze traffic to/from own security module (OR)
    (b) Analyze network traffic
        i. Install sniffer (OR)
        ii. Man-in-the-middle attack

Figure 6.15: Attack tree for the threat of information flow properties.

6.6.2 Summary

We evaluated the security of the entire system and showed how security threats are countered by security enforcing functions and mechanisms. To model the security threats against the system we made use of attack trees which provide a formal, methodical way of describing the security of systems, based on varying attacks. We examined three attacker goals and created an attack tree for each of these goals. The main goals of an attacker are to violate the safety, the liveness, and the information flow properties of the system. We analysed the attack each leaf node presented in detail and identified the conditions for the attack as well as the capabilities needed by the attacker.

The analysis indicates that the system is secure against almost all discussed attacks. An attacker can only be successful in violating the safety property of the system by attacking the eventual strong accuracy property of the failure detector. However, the attacker is not successful in preventing the execution of the consensus algorithm. Weak physical protection of system components could make the system vulnerable to attacks but in order to be efficient a large amount of system components must be attacked what makes this type of attack highly unlikely. Attention should also be paid to side-channel attacks against smartcards since all cryptographic algorithms are assumed to be vulnerable to side-channel cryptanalysis if there are no special countermeasures in the implementation. Due to the large complexity and effort to perform a side-channel attack makes this type of attack also highly unlikely.

Chapter 7

Message Omission Robust Weakest Failure Detectors

In this last chapter on message omission models, we study the impact of message omission failures on asynchronous distributed systems with crash-stop failures. We provide two different transformations for algorithms, failure detectors, and problem specifications, one of which is weakest failure detector preserving. [41]

We prove that our transformation of failure detector Ω [27] is the weakest failure detector for consensus in environments with crash-stop and permanent omission failures and a majority of correct processes.

Our results help to use the power of the well-understood crash-stop model to automatically derive solutions for the message omission model, which has recently raised interest for being noticeably applicable for security problems in distributed environments equipped with security modules such as smartcards or other trusted coprocessors [53, 54, 11].

In Section 7.1, we motivate automatic transformations and cite related work. In Section 7.2, we define our formal system model, in Section 7.3, we define our general problem and algorithm transformations, and finally in Section 7.4 we state and prove our theorems.

7.1 Motivation

Message omission failures, which have been introduced by Hadzilacos [61] and been refined by Perry and Toueg [104], put the blame of a message loss to a specific process instead of an unreliable message channel. Beyond the theoretical interest, omission models are also interesting for practical problems like they arise from the security area: Assume that some kind of trusted smartcards are disposed on untrusted processors. If these smartcards execute trusted algorithms and are able to sign messages, then it is relatively easy to restrict the power of a malicious adversary to only be able to drop messages of the trusted smartcards or to stop the smartcards themselves. Following this approach, omission models have lead to the development of reductions from security problems in the Byzantine failure model [80] such as fair-exchange [11, 54], and secure multiparty computation [53] to well-known distributed problems in the message omission model, such as consensus [32], where both process crashes and message omissions may take place. Apart from that, omission failures can model overflows of local message buffers in typical communication environments.

The message omission and crash failures are considered here in asynchronous systems. Due to classical impossibility results concerning problems as consensus [52] in asynchronous systems, following the failure detector approach [28], we augment the system with oracles that give information about failures.

The extension of failure detectors to more severe failure models than crash failures is unclear [45], because in these models failures may depend on the scheduling and on the algorithm. As it is easy to transform the message omission model into a model with only permanent omissions using standard techniques like the piggybacking of messages, we consider only permanent omissions and crashes. This means that if an omission failure occurs, then it occurs permanently. In this model, precise and simple definitions for failure detectors can easily be deduced from the ones in the crash-stop model.

To provide the permanent omission model with the benefits of a well-understood system model like the crash-stop model, we give automatic transformations for problem specifications, failure detectors, and algorithms such that algorithms designed to tolerate only crash-stop failures can be executed in permanent omission environments and use transformed failure detectors to solve transformed problems. Specifically, we give two transformations. At first, one that works in every environment, but that transforms uniform problems into problems with only limited uniformity, and at second one that works only with a majority of correct processes, but transforms uniform crash-stop problems into their uniform permanent omission counterpart. An interesting point is the fact that the transformation of the specification gives for most of the classical problems the standard specification in the message omission and crash failure model. For example, from an algorithmic solution A of the consensus problem with a failure detector $\mathcal{D}$ in the crash-stop model, we automatically get $A' = \mathit{trans}(A)$, an algorithmic solution of the consensus problem using $\mathcal{D}' = \mathit{trans}(\mathcal{D})$ in the message omission and crash failure model.

Moreover, our first transformation preserves also the "weaker than" relation [27] between failure detectors. This means that if a failure detector is a weakest failure detector for a certain (crash-stop) problem, then its transformation is a weakest failure detector for the transformed problem. We can use this to show that our transformation of failure detector Ω [27] is the weakest failure detector for (uniform) consensus in an environment with permanent omission failures and a majority of correct processes.

The problem of automatically increasing the fault-tolerance of algorithms in environments with crash-stop failures has been extensively studied before [14, 96, 40, 15]. The results of Neiger and Toueg [96], Delporte-Gallet et al. [40], and Bazzi and Neiger [15] assume in contrast to ours synchronous systems and no failure detectors. Neiger and Toueg [96] propose several transformations from crash-stop to send omission, to message omission, and to Byzantine faults. Delporte-Gallet et al. [40] transform round-based algorithms with broadcast primitives into crash-stop-, message omission-, and Byzantine-tolerant algorithms. Asynchronous systems are considered by Basu, Charron-Bost, and Toueg [14] but in the context of link failures instead of omission failures and also without failure detectors. The types of link failures that are considered by Basu, Charron-Bost, and Toueg [14] are eventually reliable and fair-lossy links. Eventually reliable links can lose a finite (but unbounded) number of messages and fair-lossy links satisfy that if infinitely many messages are sent over it, then infinitely many messages do not get lost. To show our results, we extend the system model of Basu, Charron-Bost, and Toueg [14] such that we can model omission failures, failure patterns, and failure detectors. Another definition for a system model with crash-recovery failures, omission failures, and failure detectors is given by Dolev et al. [43]. In this model, the existence of a fully connected component of processes that is completely detached from all other processes is assumed and only the processes in this component are declared to be correct.

The omission failure detector defined by Delporte-Gallet et al. [38] that can be implemented in partially synchronous models using some weak timing assumptions, is in comparison with our transformed Ω strictly stronger. However, with a correct majority, both failure detectors can easily be transformed into each other.

To the best of our knowledge, this is the first work that investigates an automatic transformation to increase the fault tolerance of distributed algorithms in asynchronous systems augmented with failure detectors.

7.2 Model

The asynchronous distributed system is assumed to consist of n distinct fully-connected processes $\Pi = \{p_1, \ldots, p_n\}$. The asynchrony of the system means, that there are no bounds on the relative process speeds and message transmission delays. To allow an easier reasoning, a discrete global clock $\mathcal{T}$ is added to the system. The system model used here is derived from that of Basu, Charron-Bost, and Toueg [14]. It has been adapted to model also failure detectors and permanent omission failures.

Algorithms

An algorithm A is defined as a vector of local algorithm modules (or simply modules) $A(\Pi) = \langle A(p_1), \ldots, A(p_n) \rangle$. Each local algorithm module $A(p_i)$ is associated with a process $p_i \in \Pi$ and defined as a deterministic infinite state automaton. The local algorithm modules can exchange messages via send and receive primitives. We assume all messages to be unique.

Failures and Failure Patterns

A failure pattern $\mathcal{F}$ is a function that maps each value t from $\mathcal{T}$ to an output value that specifies which failures have occurred up to time t during an execution of a distributed system. Such a failure pattern is totally independent of any algorithm. A crash-failure pattern

$$C : \mathcal{T} \to 2^{\Pi}$$

denotes the set of processes that have crashed up to time t ($\forall t : C(t) \subseteq C(t+1)$).

Additionally to the crash of a process, it can fail by not sending or not receiving a message. We say that it *omits* a message. The message omissions do not occur because of link failures, they model overflows of local message buffers or the behavior of a malicious adversary with control over the message flow of certain processes. It is important that for every omission, there is a process responsible for it. As we already mentioned, we consider only permanent omissions and leave the treatment of transient omissions over to the underlying asynchronous communication layer. Intuitively, a process has a permanent send omission if it always fails by not sending messages to a certain other process after a certain point in time. Analogously, a process has a permanent receive omission if it always fails by not receiving messages from a certain other process after a certain point in time. The permanent omissions are modeled via a send- and a receive-omission failure pattern:

$$O_S : \mathcal{T} \to 2^{\Pi \times \Pi} \quad \text{and} \quad O_R : \mathcal{T} \to 2^{\Pi \times \Pi}$$

If $(p_s, p_d) \in O_S(t)$, then process p_s has a permanent send-omission to process p_d after time t. If $(p_s, p_d) \in O_R(t)$, then process p_d has a permanent receive-omission to process p_s after time t. All the failure patterns defined so far can be put together to a single failure pattern $\mathcal{F} = (C, O_S, O_R)$.

With such a failure pattern, we define a process to be *correct*, if it experiences no failure at all. We assume that at least one process is correct. A process p is crash-correct ($p \in \mathit{cr.\text{-}correct}(\mathcal{F})$) in $\mathcal{F}$, if it does not crash.

A process p_d is *directly-reachable* from another process p_s in $\mathcal{F}$, if for all $t \in \mathcal{T}$, $(p_s, p_d) \notin O_S(t)$ and $(p_s, p_d) \notin O_R(t)$. A process p_d is called *reachable* from a process p_s, if p_d is directly-reachable from p_s, or if there exists a process q, such that p_d is reachable from q and q is reachable from p_s (transitive closure). If a process is reachable from some correct processes, then it is *in-connected*. Analogously, a process is *out-connected*, if some correct processes are reachable from it. If a process p is in-connected and out-connected in a failure pattern $\mathcal{F}$, then we say that p is *connected* in $\mathcal{F}$ ($p \in \mathit{connected}(\mathcal{F})$). This means that between connected processes there is always reliable communication possible. With a simple relaying algorithm, every message can eventually be delivered. Note that it is nevertheless still possible that connected processes receive messages from disconnected processes or disconnected processes receive messages from connected ones. The difference between connected and disconnected processes is that the former are able to send and to receive messages to/from correct processes and therefore are able to communicate in both directions. It is easy to see that $\mathit{crash\text{-}correct}(\mathcal{F}) \supseteq \mathit{connected}(\mathcal{F}) \supseteq \mathit{correct}(\mathcal{F})$.

We say that a failure pattern $\mathcal{F}'$ is an *omission equivalent extension* of another failure pattern $\mathcal{F}$ ($\mathcal{F} \leq_{om} \mathcal{F}'$), if the set of crash-correct processes in $\mathcal{F}$ is at all times equal to the set of connected processes in $\mathcal{F}'$ and there are no omission failures in $\mathcal{F}$.

We define an *environment* $\mathcal{E}$ to be a set of possible failure patterns. $\mathcal{E}^f_{c.s.}$ denotes the set of all failure patterns where only crash-stop faults occur and at most f processes crash. $\mathcal{E}^f_{p.o.}$ denotes the set of all failure patterns where crash-stop and permanent omission faults may occur and at most f processes are not connected (clearly, $\mathcal{E}^f_{c.s.} \subseteq \mathcal{E}^f_{p.o.}$).

Failure Detectors

A failure detector provides (possibly incorrect) information about a failure pattern [28]. Associated with each failure detector is a (possibly infinite) range $\mathcal{R}$ of values output by that failure detector. A failure detector history *FDH* with range $\mathcal{R}$ is a function from $\Pi \times \mathcal{T}$ to $\mathcal{R}$. $\mathit{FDH}(p, t)$ is the value of the failure detector module of process p at time t. A failure detector $\mathcal{D}$ is a function that maps a failure pattern $\mathcal{F}$ to a *set* of failure detector histories with range $\mathcal{R}$. $\mathcal{D}(\mathcal{F})$ denotes the set of possible failure detector histories permitted by $\mathcal{D}$ for the failure pattern $\mathcal{F}$. Note that a failure detector $\mathcal{D}$ is specified as a function of the failure pattern $\mathcal{F}$ of an execution. However, an implementation of $\mathcal{D}$ may use other aspects of the execution such as when messages are arrived and executions with the same failure pattern $\mathcal{F}$ may still have different failure detector histories. It is for this reason that we allow $\mathcal{D}(\mathcal{F})$ to be a set of failure detector histories from which the actual failure detector history for a particular execution is selected non-deterministically.

Take failure detector Ω [27] as an example. The output of the failure detector module of Ω at a process p_i is a *single* process, p_j, that p_i currently considers to be *crash-correct.* In this case, the range of output values is $\mathcal{R}_\Omega = \Pi$. For each failure pattern $\mathcal{F}$, $\Omega(\mathcal{F})$ is the set of all failure detector histories FDH_Ω with range $\mathcal{R}_\Omega$ that satisfy the following property: There is a time after which all the crash-correct processes always trust the same crash-correct process:

$$\begin{aligned} &\exists t \in \mathcal{T}, \exists p_j \in \mathit{cr.\text{-}correct}(\mathcal{F}), \\ &\forall p_i \in \mathit{cr.\text{-}correct}(\mathcal{F}), \forall t' \geq t \quad : \quad \mathit{FDH}_\Omega(p, t') = p_j \end{aligned}$$

The output of failure detector module Ω at a process p_i may change with time, i.e. p_i may trust different processes at different times. Furthermore, at any given time t, processes p_i and p_j may trust different processes.

A local algorithm module $A(p_i)$ can access the current output value of its local failure detector module using the action *queryFD*.

Histories

A local history of a local algorithm module $A(p_i)$, denoted $H[i]$, is a finite or an infinite sequence of alternating states and events of type *send*, *receive*, *queryFD*, or *internal*. We assume that there is a function *time* that assigns every event to a certain point in time and define $H[i]/_t$ to be the maximal prefix of $H[i]$ where all events have occurred before time t. A *history* H of $A(\Pi)$ is a vector of local histories $\langle H[1], H[2], \dots, H[n] \rangle$.

Reliable Links

A reliable link does not create, duplicate, or lose messages. Specifically, if there is no permanent omission between two processes and the recipient executes infinitely many receive actions, then it will eventually receive every message. We specify, that our underlying communication channels ensure reliable links.

Problem Specifications

Let Π be a set of processes and A be an algorithm. We define $\mathcal{H}(A(\Pi), \mathcal{E})$ to be the set of all tuples $(H, \mathcal{F})$ such that H is a history of $A(\Pi)$, $\mathcal{F} \in \mathcal{E}$, and H and $\mathcal{F}$ are compatible, that is crashed processes do not take any steps after the time of their crash, there are no receive-events after a permanent omission, etc. A *system* $\mathcal{S}(A(\Pi), \mathcal{E})$ of $A(\Pi)$ is a subset of $\mathcal{H}(A(\Pi), \mathcal{E})$. A *problem specification* Σ is a set of tuples of histories and failure patterns, because (permanent) omission failures are not necessarily reflected in a history (e.g., if a process sends no messages). A system $\mathcal{S}$ satisfies a problem specification Σ, if $\mathcal{S} \subseteq \Sigma$. We say that an algorithm A satisfies a problem specification Σ in environment $\mathcal{E}$, if $\mathcal{H}(A(\Pi), \mathcal{E}) \subseteq \Sigma$.

Take consensus as an example (see Figure 7.4): It is specified by making statements about some variables *propose* and *decide* in the states of a history (e.g. the value of *decide* has eventually to be equal at all (crash-)correct processes). This can be expressed as the set of all tuples $(H, \mathcal{F})$ where there exists a time t and a value v, such that for all $p_i \in \textit{cr.-correct}(\mathcal{F})$, there exists an event e in $H[i]$ with $\textit{time}(e) \leq t$ and for all states s after event e, the value of the variable *decide* in s is v.

7.3 From Crash-Stop to Permanent Omission

We will give here two transformations: one general transformation for all environments, where we provide only restricted guarantees for disconnected processes, and one for environments where less than half of the processes may not be connected, where we are able to provide for all processes the same guarantees as for the crash-stop case.

To improve the fault-tolerance of algorithms, we simulate a single state of the original algorithm with several states of the simulation algorithm. For these additional states, we *augment* the original states with additional variables. Since an event of the simulation algorithm may lead to a state where only the augmentation variables change, the sequence of the original variables may *stutter*. We call a local history $H'[i]$ a stuttered and augmented extension of a history $H[i]$ ($H[i] \leq_{sa} H'[i]$), if $H[i]$ and $H'[i]$ differ only in the value of the augmentation variables and some additional states caused by differences in these variables (in particular, $H[i] \leq_{sa} H[i]$ for all $H[i]$). If $H[i] \leq_{sa} H'[i]$ for all $p_i \in \Pi$, we write $H \leq_{sa} H'$. We say that a problem specification Σ is closed under stuttering and augmentation, if $(H, \mathcal{F}) \in \Sigma$ and $H \leq_{sa} H'$ implies that $(H', \mathcal{F})$ is also in Σ. Most problems satisfy this natural closure property (e.g. consensus).

7.3.1 The General Transformation

Transformation of Problem Specifications

To transform a problem specification, we first show a transformation of a tuple of a trace and a failure pattern. Based on this transformation, we transform a whole problem specification. The intuition behind this transformation is that we demand only something from processes as long as they are connected. After their disconnection, processes may behave arbitrary. More formally, let $t_{c.s.}(i)$ be the time at which process p_i crashes in $\mathcal{F}$ ($t_{c.s.}(i) = \infty$, if p_i never crashes). Analogously, let $t'_{p.o.}(i)$ be the time at which process p_i becomes disconnected in $\mathcal{F}'$ ($t'_{p.o.}(i) = \infty$, if p_i never becomes disconnected). Then:

$$(H', \mathcal{F}') \in \mathit{trans}((H, \mathcal{F})) \quad :\Leftrightarrow \quad \forall p_i \in \Pi : \; H[i]/_{t_{c.s.}(i)} \leq_{sa} H'[i]/_{t'_{p.o.}(i)}$$

and for a whole problem specification:

$$\mathit{trans}(\Sigma) := \{(H', \mathcal{F}') \mid (H', \mathcal{F}') \in \mathit{trans}((H, \mathcal{F})) \wedge (H, \mathcal{F}) \in \Sigma\}$$

A transformation of non-uniform consensus, where properties of certain propose- and decision-variables of (crash-)correct processes are specified would lead to a specification where the same properties are ensured for the states of connected processes, because only histories with the same states (disregarding the augmentation variables) are allowed in the transformation at this processes (see Figure 7.4). We also take the states of processes *before* they become disconnected into account, because they (e.g. their initial states for the propose variables) may also have an influence on the fulfillment of a problem specification, although they are after their disconnection not allowed to have this influence anymore. Since we impose no restriction on the behavior of processes after their disconnection, the transformed problem specification allows them to decide a value that was never proposed (although our transformation algorithms guarantee that this will not happen).

A transformation of uniform consensus leads to a problem specification where the uniform agreement is only demanded for processes before their time of disconnection. This means that it is allowed that after a partitioning of the network, the processes in the different network partitions come to different decision values. Another transformation, in which uniform consensus remains truly uniform is given in Section 7.3.2.

Transformation of Failure Detector Specifications

We allow all failure detector histories for a failure pattern $\mathcal{F}$ in $\mathit{trans}(\mathcal{D})$ that are allowed in the crash-stop version $\mathcal{F}'$ of $\mathcal{F}$ in $\mathcal{D}$:

$$\mathit{trans}(\mathcal{D})(\mathcal{F}) := \bigcup_{\mathcal{F}'} \{\mathcal{D}(\mathcal{F}') \mid \mathcal{F}' \leq_{om} \mathcal{F}\}$$

Consider failure detector Ω [27]. Ω outputs only failure detector histories that eventually provide the same crash-correct leader at all crash-correct processes. Then, $\mathit{trans}(\Omega)$ outputs these failure detector histories if and only if they provide a *connected* common leader at all *connected* processes.

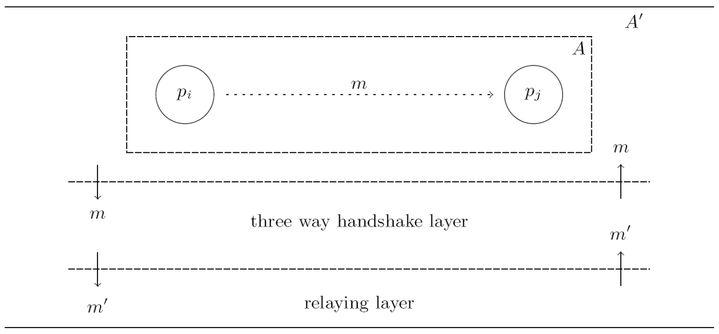

Figure 7.1: Additional Communication Layers

Transformation of Algorithms

In our algorithm transformation, we add new communication layers such that some of the omission failures in the system become transparent to the algorithm (see Figure 7.1). We transform a given algorithm A into another algorithm $A' = \mathit{trans}(A)$ in two steps:

- In the first step, we remove the send and receive actions from A and simulate them with a *three-way-handshake (3wh) algorithm*. The algorithm is described in Figure 7.2. The idea of the 3wh-algorithm is to substitute every send-action with an exchange of three messages. This means that to send a message to a certain process, it is necessary for a process to be able to send *and* to receive messages from it. Moreover, while the communication between connected processes is still possible, processes that are only in-connected or only out-connected (and not both) become totally disconnected. Hence, we eliminate influences of disconnected processes not existing in the crash-stop case.

- Then, in the second step, we remove the send and receive actions from the three way handshake algorithm and simulate them with a *relaying algorithm*. The idea of the relay algorithm is to relay every message to all other processes, such that they relay it again and all connected processes can communicate with each other, despite the fact that they are not directly-reachable. It is similar to other algorithms in the literature [113]. Its detailed description can be found in Figure 7.3.

To execute the simulation algorithms in parallel with the actions from A, we add some new (augmentation) variables to the set of variables in the states of A. Whenever a step of the simulation algorithms is executed, the state of the original variables in A remains untouched and only the new variables change their values. Whenever a process queries a local failure detector module $\mathcal{D}(p_i)$, we translate it to a query on $\mathit{trans}(\mathcal{D})(p_i)$. The relaying layer overlays the network with the best possible communication graph and the 3wh-layer on top of it cuts the unidirectional edges from this graph.

7.3.2 The Transformation for $n > 2f$

If only less than a majority of the processes are disconnected ($n > 2f$), then we only need to adapt the problem specification to the failure patterns of the new environment. We indicate this adaptation of a problem specification with the index *p.o.* and specify it in the following way:

$$\Sigma_{p.o.} := \{(H, \mathcal{F}) \mid \exists (H, \mathcal{F}') \in \Sigma \ \wedge \ \mathcal{F}' \leq_{om} \mathcal{F}\}$$

```
Algorithm 3wh
procedure 3wh-send($m, p_j$)
    relay-send($[1, m], p_j$);

procedure 3wh-receive($m$)
    relay-receive($[l, m']$);
    if ($l = 1$) then relay-send($[2, m']$, sender($[l, m']$)); $m := \bot$;
    elseif ($l = 2$) then relay-send($[3, m']$, sender($[l, m']$)); $m := \bot$;
    elseif ($l = 3$) then $m := m'$;
    elseif $[l, m'] = \bot$ then $m := \bot$;
```

Figure 7.2: The Three Way Handshake Algorithm for Process p_i.

```
Algorithm Relay
procedure init
    $relayed_i := \emptyset$; $delivered_i := \emptyset$;

procedure relay-send($m, p_j$)
    for $k := 1$ to $n$ do
        send($[m, p_j], p_k$);
    $relayed_i := relayed_i \cup \{[m, p_j]\}$;

procedure relay-receive($m$)
    receive($[m', p_k]$);
    if ($[m', p_k] = \bot$) then $m := \bot$;
    elseif ($k = i$) and ($m' \notin delivered_i$) then
        $m := m'$; $delivered_i := delivered_i \cup \{m'\}$;
    elseif ($k \neq i$) and ($[m', p_k] \notin relayed_i$) then
        for $l := 1$ to $n$ do
            send($[m', p_k], p_l$);
        $relayed_i := relayed_i \cup \{[m', p_k]\}$; $m := \bot$;
```

Figure 7.3: The Relaying Algorithm for Process p_i.

	Consensus	*trans*(Consensus)	Consensus$_{p.o.}$
Validity:	The decided value of every process must have been proposed.	The decided value of every *connected* process must have been proposed.	The decided value of every process must have been proposed.
Non-Uniform Agreement:	No two *cr.-correct* processes decide differently.	No two *connected* processes decide differently.	No two *connected* processes decide differently.
Uniform Agreement:	No two processes decide differently.	No two processes decide differently *before their disconnection.*	No two processes decide differently.
Termination:	Every *cr.-correct* process eventually decides.	Every *connected* process eventually decides.	Every *connected* process eventually decides.

Figure 7.4: Transformations of the Consensus Problem

If we adapt consensus to omission failures, then we get Consensus$_{p.o.}$ as in Figure 7.4. The failure detector specifications can be transformed as in Section 7.3.1. The algorithm transformation *trans*$_2$ works similar as in the previous section, but we add an additional *two-way-handshake (2wh) layer* between the relaying layer and the 3wh layer. The algorithm is described in Figure 7.5 and is similar to an algorithm in the literature [14]. The idea of the algorithm is to broadcast every message to all other processes and to block until $f+1$ processes have acknowledged the message. In this way, disconnected processes block forever (since they receive less than $f+1$ acknowledgements) and connected processes can continue. Thus, we emulate a crash-stop environment.

7.4 Results

In our first theorem, we show that for any algorithm A, for any failure detector $\mathcal{D}$, and for any problem specification Σ, *trans*(A) using *trans*$(\mathcal{D})$ solves *trans*(Σ) in a permanent omission environment if and only if A using $\mathcal{D}$ solves Σ in a crash-stop environment. This theorem does not only show that our transformation works, it furthermore ensures that we do not transform to a trivial problem specification, but to an equivalent one, since we prove both directions.

Theorem 7.1. *Let Σ be a problem specification closed under stuttering and augmentation. Then, if A is an algorithm using a failure detector $\mathcal{D}$ and $A' = trans(A)$ is the transformation of A using trans$(\mathcal{D})$, it holds that:*

$$\begin{aligned} \forall f \text{ with } 0 \le f \le n: \quad & (\mathcal{H}(A(\Pi), \mathcal{E}^f_{c.s.}) \subseteq \Sigma \\ \Leftrightarrow \quad & (\mathcal{H}(A'(\Pi), \mathcal{E}^f_{p.o.}) \subseteq trans(\Sigma) \end{aligned}$$

Proof. **(Sketch)** Due to lack of space, we only sketch the proof of the theorem here. The detailed proof can be found elsewhere [37]. The proof is divided up into two parts. Let $\mathcal{S}_{c.s.} := (\mathcal{H}(A(\Pi), \mathcal{E}^f_{c.s.})$ and $\mathcal{S}_{p.o.} := (\mathcal{H}(A'(\Pi), \mathcal{E}^f_{p.o.})$ and assume that $A' = trans(A)$.

"$\Rightarrow$": Assume that $\mathcal{S}_{c.s.} \subseteq \Sigma$. By constructing for a given $(H, \mathcal{F})$ in $\mathcal{S}_{p.o.}$ a tuple $(H', \mathcal{F}')$ in $\mathcal{S}_{c.s.}$ with $(H, \mathcal{F}) \in trans((H', \mathcal{F}'))$, we can show that $\mathcal{S}_{p.o.} \subseteq trans(\mathcal{S}_{c.s.})$. In

```
Algorithm 2wh
1:  procedure init
2:      $received_i := \emptyset$; $Ack_i := 0$;
3:
4:  procedure 2wh-send$(m, p_j)$
5:      relay-send$([m, p_j, ONE], p_k)$ to all other $p_k$; $Ack_i := 1$;
6:      while $(Ack_i \leq f)$ do
7:          relay-receive$([m', p_k, num])$);
8:          if $(num = TWO)$ and $(m' = m)$ and $(k = j)$ then inc$(Ack_i)$;
9:          elseif $(num = ONE)$ then add $[m', p_k, num]$ to $received_i$;
10:
11: procedure 2wh-receive$(m)$)
12:     $m := \perp$; relay-receive$(m')$;
13:     if $(m' \neq \perp)$ then add $m'$ to $received_i$;
14:     if $([m'', p_k, ONE] \in received_i)$ for any $m'', p_k$ then
15:         relay-send$([m'', p_k, TWO], sender([m'', p_k, ONE]))$;
16:         if $(k = i)$ then $m := m''$;
```

Figure 7.5: The Two Way Handshake Algorithm for Process p_i.

this construction, we remove the added communication layers from H and use the properties of our two send-primitives to prove the reliability of the links in H'. We ensure "No Loss" with the relaying algorithm and "No Creation" with the three way handshake algorithm. As we know from the definition of *trans*, that $\mathit{trans}(\mathcal{S}_{c.s.}) \subseteq \mathit{trans}(\Sigma)$, we can conclude that $\mathcal{S}_{p.o.} \subseteq \mathit{trans}(\Sigma)$.

"⇐": Assume that $\mathcal{S}_{p.o.} \subseteq \mathit{trans}(\Sigma)$ and $(H, \mathcal{F}) \in \mathcal{S}_{c.s.}$. We then build a new history H' from H and simulate all links according to the specification of the three-way-handshake and the relay algorithm such that $(H', \mathcal{F}) \in \mathit{trans}((H, \mathcal{F}))$ and $(H', \mathcal{F}) \in \mathcal{S}_{p.o.} \subseteq \mathit{trans}(\Sigma)$ ($\mathcal{F} \in \mathcal{E}^f_{c.s.}$ implies that $\mathcal{F}$ is in $\mathcal{E}^f_{p.o}$). This means that there exists a $(H'', \mathcal{F}'') \in \Sigma$, with $(H', \mathcal{F}) \in \mathit{trans}((H'', \mathcal{F}''))$.

Since in both, $\mathcal{F}''$ and $\mathcal{F}$ occur only crash failures, $\mathcal{F}'' = \mathcal{F}$ and therefore for all p_i, $H''[i] \leq_{sa} H'[i]$. Together with the fact that Σ is closed under stuttering and augmentation, we can conclude that $(H', \mathcal{F}) \in \Sigma$. H' and H differ only in the augmentation variables that are not relevant for the fulfillment of $\mathit{trans}(\Sigma)$ and therefore: $(H, \mathcal{F}) \in \Sigma$.

□

Proof. We divide up the proof into two parts. Let $\mathcal{S}_{c.s.} := (\mathcal{H}(A(\Pi), \mathcal{E}^f_{c.s.})$ and $\mathcal{S}_{p.o.} := (\mathcal{H}(A'(\Pi), \mathcal{E}^f_{p.o.})$ and assume that $A' = \mathit{trans}(A)$.

"⇒": Assume that $\mathcal{S}_{c.s.} \subseteq \Sigma$. By constructing for a given $(H, \mathcal{F})$ in $\mathcal{S}_{p.o.}$ a tuple $(H', \mathcal{F}')$ in $\mathcal{S}_{c.s.}$ with $(H, \mathcal{F}) \in \mathit{trans}((H', \mathcal{F}'))$, we can show that $\mathcal{S}_{p.o.} \subseteq \mathit{trans}(\mathcal{S}_{c.s.})$ (Proposition 7.2). In this construction, we remove the added communication layers from H and use the properties of our two send-primitves to prove the reliability of the links in H'. We ensure "No Loss" with the relaying algorithm and "No Creation" with the three way handshake algorithm. As we know from the definition of *trans*, that $\mathit{trans}(\mathcal{S}_{c.s.}) \subseteq \mathit{trans}(\Sigma)$, we can conclude that $\mathcal{S}_{p.o.} \subseteq \mathit{trans}(\Sigma)$.

"⇐": Assume that $\mathcal{S}_{p.o.} \subseteq \mathit{trans}(\Sigma)$. We construct $(H', \mathcal{F}')$ for all $(H, \mathcal{F})$ in $\mathcal{S}_{c.s.}$, such that $(H', \mathcal{F}')$ is in $\mathcal{S}_{p.o.} \subseteq \mathit{trans}(\Sigma)$. We can use this to prove that $\mathcal{S}_{c.s.} \subseteq \Sigma$ (Proposition 7.10).

$\square$

Proposition 7.2. $\mathcal{S}_{p.o.} \subseteq \mathit{trans}(\mathcal{S}_{c.s.})$

Proof. The proposition is equivalent to

$$(H, \mathcal{F}) \in \mathcal{S}_{p.o.} \Rightarrow (H, \mathcal{F}) \in \mathit{trans}(\mathcal{S}_{c.s.})$$

From the definition of *trans* follows:

$$\begin{aligned} (H, \mathcal{F}) \in \mathcal{S}_{p.o.} \quad &\Rightarrow \quad \exists (H', \mathcal{F}') \in \mathcal{S}_{c.s.} : \\ \forall p_i \in \Pi \quad &: \quad H'[i]/_{t'_{c.s.}(i)} \leq_{sa} H[i]/_{t_{p.o.}(i)} \end{aligned} \tag{7.1}$$

We will in the following construct a new history H' and a failure pattern $\mathcal{F}'$ from H and $\mathcal{F}$ which satisfy equation (7.1):

(a) At first, we undo step 2 of the transformation and remove the variables, additional states, and events of the relaying algorithm from H. This means, that every time a relay-send or relay-receive event in H occurs, this event is substituted by an send/receive event of the underlying communication channel. We let the inserted events take place at the time when the relay events have been completed (since a process may take several steps to accomplish the relaying task). We call the intermediate history we get after this H_1.

(b) Then, we undo step 1 and remove the variables, additional states, and events of the three way handshake algorithm from H_1 (in the same way as above). We call this intermediate history H_2.

(c) After that, we construct $\mathcal{F}'$, such that $\mathcal{F}' \leq_{om} \mathcal{F}$. To build H' from H_2, we substitute every query on a failure detector $\mathit{trans}(\mathcal{D})$ in H_2 with a query on $\mathcal{D}$ in H' and remove all states and events for every process p_i that occur after the time when p_i crashes in $\mathcal{F}'$.

The schedule of the construction is illustrated in Figure 7.6. From the construction of H' and $\mathcal{F}'$ it is clear, that $\forall p_i \in \Pi : \; H'[i]/_{t'_{c.s.}(i)} \leq_{sa} H[i]/_{t_{p.o.}(i)}$. It remains to show, that $(H', \mathcal{F}') \in \mathcal{S}_{c.s.}$. This means, that at most f processes crash in $\mathcal{F}'$ (Lemma 7.3), H' is a history of $A(\Pi)$ using $\mathcal{D}$ (Lemma 7.4), and all links in H' are reliable according to $\mathcal{F}'$ (Lemma 7.5).

$$\begin{tikzcd}[column sep=huge] H \arrow[r, "\substack{\text{(a): undo step 2}\\ \text{(the relaying)}}"'] & H_1 \arrow[r, "\substack{\text{(b): undo step 1}\\ \text{(the 3wh)}}"'] & H_2 \arrow[r, "\substack{\text{(c): crash not}\\ \text{connected processes}}"'] & H' \end{tikzcd}$$

Figure 7.6: Construction of H'

$\square$

Lemma 7.3. *At most f processes crash in $\mathcal{F}'$.*

Proof. Follows immediately from (c). $\square$

Lemma 7.4. *H' is a history of $A(\Pi)$ using $\mathcal{D}$.*

Proof. All events and states are from $A(\Pi)$, because all additional events and states have been removed. If algorithm A makes use of a failure detector $\mathcal{D}$, then $\mathit{trans}(\mathcal{D})(\mathcal{F}) = \mathcal{D}(\mathcal{F}')$ (Since $\mathcal{F}' \leq_{om} \mathcal{F}$). □

Lemma 7.5. *All links in H' are reliable according to $\mathcal{F}'$.*

Proof. We have to show the three properties of reliable links, namely: No Creation (Lemma 7.7), No Duplication (Lemma 7.8), and No Loss (Lemma 7.9). □

To prove lemma 7.7, we first need to show the auxiliary lemma 7.6:

Lemma 7.6. *Let t_s be the time a send event from $A(p_i)$ to $A(p_j)$ in H_2 occurs, t_r be the time of the corresponding receive event in H_2, t_j be the time when p_j becomes disconnected in $\mathcal{F}$, and t_i be the time when p_i become disconnected in $\mathcal{F}$. Then:*

$$t_s \geq t_i \Rightarrow t_r \geq t_j$$

Proof. In the following, when we write $t_{nr(\mathcal{F},p,q)}$, we mean the point in time when process p is not longer reachable from process q in $\mathcal{F}$ (for any p, q, and $\mathcal{F}$).

The above lemma is equivalent to: $t_r < t_j$ implies $t_s < t_i$. At first, we observe that $t_s < t_r$. Assume $t_r < t_j$. Since $A(p_j)$ receives the message, we can conclude:

$$t_{nr(\mathcal{F},p_j,p_i)} \quad > \quad t_r > t_s \tag{7.2}$$

Since the in H_2 removed 3wh-algorithm is only allowed to 3wh-deliver messages after having received a $[3, m]$ message (line 12 in Figure 7.2), which is only sent from a process after having on his part received a $[2, m]$ message (line 11), we are sure that after the 3wh-send event, $A(p_i)$ was able to receive the $[2, m]$ message from $A(p_j)$ and therefore:

$$t_{nr(\mathcal{F},p_i,p_j)} \quad > \quad t_s \tag{7.3}$$

From the definition of connected follows:

$$\exists c \in \mathit{correct}(\mathcal{F}), t_{nr(\mathcal{F},c,p_j)} \geq t_j > t_r > t_s \tag{7.4}$$

$$\exists c' \in \mathit{correct}(\mathcal{F}), t_{nr(\mathcal{F},p_j,c')} \geq t_j > t_r > t_s \tag{7.5}$$

If we put all paths together, we have:

$$\text{with (7.2) \& (7.4)} \quad : \quad \exists c \in \mathit{correct}(\mathcal{F}), t_{nr(\mathcal{F},c,p_i)} > t_s \tag{7.6}$$

$$\text{with (7.3) \& (7.5)} \quad : \quad \exists c' \in \mathit{correct}(\mathcal{F}), t_{nr(\mathcal{F},p_i,c')} > t_s \tag{7.7}$$

Equations (7.6) and (7.7) imply $t_i > t_s$. □

Lemma 7.7. *(No Creation in H'.) For all messages m, if p_j receives m from p_i in H', then p_i sends m to p_j in H'.*

Proof. We know, that there is no creation in H. In our construction, send events of the same layer can only decrease in the local history of crashed processes in step (c) (after the time of their crash). But since Lemma 7.6 shows that messages that are sent from a process that is already disconnected in $\mathcal{F}$ (and therefore crashed in $\mathcal{F}'$) can only be received by processes that are already disconnected too, the corresponding receive events also get lost in H'. □

Lemma 7.8. *(No Duplication in H'.) For all messages m: p_j receives m from p_i at most once.*

Proof. In the 3wh-algorithm, no message is delivered more than once and in the relay-algorithm, every message received is remembered in a variable $\mathit{delivered}_i$ (lines 12-13 in Figure 7.3). □

Lemma 7.9. *(No Loss in H' according to $\mathcal{F}'$.) For all messages m, if p_i sends m to p_j and p_j executes receive actions infinitely often, then p_j receives m from p_i.*

Proof. In the removed relaying algorithm, after every relay-send event, the message m is relayed by $A(p_i)$ to all other processes (lines 5-6 in Figure 7.3). If a connected process (in $\mathcal{F}$) receives such a relayed message, it checks in lines 12-13 whether it is the recipient and has not yet delivered it (and relay-delivers m in this case). Otherwise, it propagates m further to all other processes (lines 14-16).

Since p_i is at the time of the in step (a) in H_1 inserted send-event out-connected in $\mathcal{F}$ (otherwise, p_i would have already crashed in $\mathcal{F}'$), there is a path of directly-reachable connected processes to a (totally) correct process in $\mathcal{F}$. A correct process will receive m and relay it (possibly indirectly) to $A(p_j)$, since p_j is in-connected in $\mathcal{F}$ (because it takes infinitely many steps in $(H', \mathcal{F}')$). □

Proposition 7.10. $\mathcal{S}_{c.s.} \subseteq \Sigma$

Proof. Assume $(H, \mathcal{F}) \in \mathcal{S}_{c.s.}$. We then build a new history H' from H and simulate all links according to the specification of the three-way-handshake and the relay algorithm such that $(H', \mathcal{F}) \in \mathit{trans}((H, \mathcal{F}))$ and $(H', \mathcal{F}) \in \mathcal{S}_{p.o.} \subseteq \mathit{trans}(\Sigma)$ ($\mathcal{F} \in \mathcal{E}^f_{c.s.}$ implies that $\mathcal{F} \in \mathcal{E}^f_{p.o}$). This means, that there exists a $(H'', \mathcal{F}'') \in \Sigma$, with $(H', \mathcal{F}) \in \mathit{trans}((H'', \mathcal{F}''))$.

Since in both, $\mathcal{F}''$ and $\mathcal{F}$ occur only crash failures, $\mathcal{F}'' = \mathcal{F}$ and therefore for all p_i, $H''[i] \leq_{sa} H'[i]$. Together with the fact that Σ is closed under stuttering and augmentation, we can conclude that $(H', \mathcal{F}) \in \Sigma$. H' and H differ only in the augmentation variables that are not relevant for the fulfillment of $\mathit{trans}(\Sigma)$, therefore: $(H, \mathcal{F}) \in \Sigma$.

□

Our second theorem shows, that with a majority of connected processes ($n > 2f$), trans_2 can be used to solve the adaptation of a problem to the message omission model.

Theorem 7.11. *If A is an algorithm using a failure detector $\mathcal{D}$ and $A' = \mathit{trans}_2(A)$ is the transformation of A using $\mathit{trans}_2(\mathcal{D})$ and Σ is closed under stuttering and augmentation, then it holds that:*

$$\begin{aligned} \forall f \text{ with } f < n/2 \quad &: \quad (\mathcal{H}(A(\Pi), \mathcal{E}^f_{c.s.}) \subseteq \Sigma \\ &\Rightarrow \quad (\mathcal{H}(A'(\Pi), \mathcal{E}^f_{p.o.}) \subseteq \Sigma_{p.o.} \end{aligned}$$

Proof. **(Sketch)** Due to lack of space, we only sketch the proof of the theorem here. The detailed proof can be found elsewhere [37]. Let $\mathcal{S}_{c.s.} := (\mathcal{H}(A(\Pi), \mathcal{E}^f_{c.s.})$ and $\mathcal{S}_{p.o.} := (\mathcal{H}(A'(\Pi), \mathcal{E}^f_{p.o.})$ and assume that $A' = \mathit{trans}(A)$. It is sufficient to show, that

$$\forall (H, \mathcal{F}) \in \mathcal{S}_{p.o.}, \exists (H', \mathcal{F}') \in \mathcal{S}_{c.s.} : \ (H' \leq_{sa} H) \wedge (\mathcal{F}' \leq_{om} \mathcal{F}) \tag{7.8}$$

To show this, we construct $(H', \mathcal{F}') \in \mathcal{S}_{c.s.}$ for a given $(H, \mathcal{F}) \in \mathcal{S}_{p.o.}$ in the following way: We first remove the variables, events, and states of the relay-algorithm, then remove the same for the 2wh-algorithm, and then remove the 3wh-algorithm to get H'. $\mathcal{F}'$ is a failure pattern, such that $\mathcal{F}' \leq_{om} \mathcal{F}$. We need to show, that $(H', \mathcal{F}')$ fulfills the properties of equation 7.9. From the construction it is clear, that $H' \leq_{sa} H$ and $\mathcal{F}' \leq_{om} \mathcal{F}$. It remains to show, that $(H', \mathcal{F}') \in \mathcal{S}_{c.s.}$. This means, that at most f processes crash in $\mathcal{F}'$, H' is a history of $A(\Pi)$ using $\mathcal{D}$, all links are reliable in $(H', \mathcal{F}')$, and H' and $\mathcal{F}'$ are compatible. Here we can use the properties of the 2wh-algorithm to ensure that a process that is crashed in $\mathcal{F}'$ takes no steps in H' after the time of its crash. □

Proof. Let $\mathcal{S}_{c.s.} := (\mathcal{H}(A(\Pi), \mathcal{E}^f_{c.s.})$ and $\mathcal{S}_{p.o.} := (\mathcal{H}(A'(\Pi), \mathcal{E}^f_{p.o.})$ and assume that $A' = \mathit{trans}(A)$. It is sufficient to show, that

$$\begin{aligned} &\forall (H, \mathcal{F}) \in \mathcal{S}_{p.o.}, \exists (H', \mathcal{F}') \in \mathcal{S}_{c.s.} : \\ &\qquad (H' \leq_{sa} H) \wedge (\mathcal{F}' \leq_{om} \mathcal{F}) \end{aligned} \tag{7.9}$$

To show this, we construct $(H', \mathcal{F}') \in \mathcal{S}_{c.s.}$ for a given $(H, \mathcal{F}) \in \mathcal{S}_{p.o.}$ in the following way: We first remove the variables, events, and states of the relay-algorithm, then remove the same for the 2wh-algorithm, and then remove the 3wh-algorithm to get H'. $\mathcal{F}'$ is a failure pattern, such that $\mathcal{F}' \leq_{om} \mathcal{F}$. We need to show, that $(H', \mathcal{F}')$ fulfills the properties of equation 7.9. From the construction it is clear, that $H' \leq_{sa} H$ and $\mathcal{F}' \leq_{om} \mathcal{F}$. It remains to show, that $(H', \mathcal{F}') \in \mathcal{S}_{c.s.}$. This means, that at most f processes crash in $\mathcal{F}'$ (Lemma 7.12), H' is a history of $A(\Pi)$ using $\mathcal{D}$ (Lemma 7.13), all links are reliable in $(H', \mathcal{F}')$ (Lemma 7.18), and H' and $\mathcal{F}'$ are compatible (Lemma 7.17). □

Lemma 7.12. *At most t processes crash in $\mathcal{F}'$.*

Proof. Follows immediately from $\mathcal{F}' \leq_{om} \mathcal{F}$. □

Lemma 7.13. *H' is a history of $A(\Pi)$ using $\mathcal{D}$.*

Proof. All events and states are from $A(\Pi)$, because all additional events and states have been removed. If algorithm A makes use of a failure detector $\mathcal{D}$, then $\mathit{trans}(\mathcal{D})(\mathcal{F}) = \mathcal{D}(\mathcal{F}')$ (Since $\mathcal{F}' \leq_{om} \mathcal{F}$). □

Lemma 7.14. *Connected processes take infinitely many steps.*

Proof. The only possibility for a process to block is in line 7 of the 2wh-algorithm in Figure 7.5. Since $n > 2f$, even after the disconnection of all f possibly faulty processes, every connected process receives acknowledgements from $n - f > f$ connected processes and therefore never blocks in line 7. □

Lemma 7.15. *Every process 2wh-sends at most one message after its disconnection.*

Proof. If p_i is disconnected after some time, it either does not receive messages from connected processes or connected processes do not receive messages from it. If it does not receive messages from connected processes, then after a 2wh-send event, it receives at most f acknowledgements (from the disconnected ones) and therefore waits forever in line 7 of the 2wh-algorithm in Figure 7.5. If the connected processes do not receive messages from it and p_i 2wh-sends a message, also at most f processes will receive the ONE-message and answer with a TWO-message. Therefore, process p_i will block forever in line 7. □

Lemma 7.16. *The state of a process in H' does not change after its disconnection.*

Proof. With the 3wh-algorithm, we can ensure that a process does not receive messages from connected processes (Lemma 7.6). With Lemma 7.15, no process sends more than one message after its disconnection (and this message is not sufficient for a 3wh). Therefore, this send event is not visible to other processes and the internal state of a disconnected process cannot be influenced after its disconnection. □

Lemma 7.17. *H' and $\mathcal{F}'$ are compatible.*

Proof. We show, that every connected process takes infinitely many steps (Lemma 7.14), and that the state of a process after its disconnection does not change anymore in H' (Lemma 7.16). □

Lemma 7.18. *All links in H' are reliable according to $\mathcal{F}'$.*

Proof. We have to show the three properties of reliable links, namely: No Creation (Lemma 7.19), No Duplication (Lemma 7.20), and No Loss (Lemma 7.21). □

Lemma 7.19. *No Creation in H'.*

Proof. There is no loss in H and the send events in the same layer never decrease. □

Lemma 7.20. *No Duplication in H'.*

Proof. In the relay- and the 3wh-handshake algorithm, there is no duplication (Lemma 7.8). In the 2wh-algorithm, only one ONE message with the correct id is sent for every 2wh-send. □

Lemma 7.21. *No Loss in H'.*

Proof. We know from Lemma 7.9, that there is no loss between connected processes without the 2wh-algorithm. With Lemma 7.14, we know connected processes take infinitely many steps and make therefore infinitely many receive actions. It remains to show, that disconnected processes stop sending and receiving messages after their disconnection (Lemma 7.16). □

Weakest Failure Detectors

A failure detector [27] is a weakest failure detector for a problem specification Σ in environment $\mathcal{E}$, if it is necessary and sufficient. Sufficient means, that there exists an algorithm using this failure detector that satisfies Σ in $\mathcal{E}$, whereas necessary means, that every other sufficient failure detector is reducible to it. A failure detector $\mathcal{D}$ is reducible to another failure detector $\mathcal{D}'$, if there exists a transformation algorithm $T_{\mathcal{D}\to\mathcal{D}'}$, such that for every tuple $(H, \mathcal{F}) \in \mathcal{H}(T_{\mathcal{D}\to\mathcal{D}'}(\Pi), \mathcal{E})$, H is equivalent to a failure detector history *FDH* in $\mathcal{D}'(\mathcal{F})$. We call the problem specification that arises in emulating $\mathcal{D}'$, *Probl*$(\mathcal{D}')$. In the following theorem, we show that *trans* preserves the weakest failure detector property for non-uniform[1] failure detectors.

Theorem 7.22. *For all f with $1 \leq f \leq n$: If a non-uniform failure detector $\mathcal{D}$ is a weakest failure detector for Σ in $\mathcal{E}^f_{c.s.}$ and Σ is closed under stuttering and augmentation, then trans$(\mathcal{D})$ is a weakest failure detector for trans(Σ) in $\mathcal{E}^f_{p.o.}$.*

[1]A non-uniform failure detector $\mathcal{D}$ outputs always the same set of histories for two failure patterns $\mathcal{F}$ and $\mathcal{F}'$ in which *correct*$(\mathcal{F})$ = *correct*$(\mathcal{F}')$ (i.e. $\mathcal{D}(\mathcal{F}) = \mathcal{D}(\mathcal{F}')$).

Proof. If $\mathcal{D}$ is a weakest failure detector for Σ in $\mathcal{E}^f_{c.s.}$, then *trans*($\mathcal{D}$) is sufficient for *trans*(Σ) in $\mathcal{E}^f_{p.o.}$ (Theorem 7.1). It remains to show that *trans*($\mathcal{D}$) is also necessary.

Assume a failure detector $\mathcal{D}'$ is sufficient for *trans*(Σ) in $\mathcal{E}^f_{p.o.}$. Clearly, $\Sigma \subseteq$ *trans*(Σ) (since $H \leq_{sa} H$ for all H). Therefore, $\mathcal{D}'$ is sufficient for Σ in $\mathcal{E}^f_{c.s.}$, and moreover, $\mathcal{D}'$ is reducible to $\mathcal{D}$ in $\mathcal{E}^f_{c.s.}$ (since $\mathcal{D}$ is a weakest failure detector for Σ in $\mathcal{E}^f_{c.s.}$). This means that it is possible to emulate $\mathcal{D}$ using $\mathcal{D}'$ (i.e. a problem specification *Probl*($\mathcal{D}$) that is equivalent to $\mathcal{D}$). If the reduction algorithm is $T_{\mathcal{D}'\rightarrow\mathcal{D}}$, then *trans*($T_{\mathcal{D}'\rightarrow\mathcal{D}}$) using *trans*($\mathcal{D}'$) emulates *trans*(*Probl*($\mathcal{D}$)) in $\mathcal{E}^f_{p.o.}$ (Theorem 7.1) and since $\mathcal{D}$ is non-uniform, the transformation of the problem specification, *trans*(*Probl*($\mathcal{D}$)) is equivalent to the transformation of the failure detector *trans*($\mathcal{D}$) (*trans* does not change the meaning of *Probl*($\mathcal{D}$) since only the states of connected processes matter). Therefore, $\mathcal{D}'$ is reducible to *trans*($\mathcal{D}$) in $\mathcal{E}^f_{p.o.}$. □

With Theorem 7.1, 7.11, and 7.22 we are able to show, the following:

Theorem 7.23. *trans*(Ω) *is a weakest failure detector for uniform Consensus*$_{p.o.}$ *with a majority of correct processes.*

Proof. Since we know, that Ω is a weakest failure detector for non-uniform Consensus [27] and Ω is clearly non-uniform, together with Theorem 7.22, *trans*(Ω) is a weakest failure detector for non-uniform *trans*(Consensus). Since non-uniform *trans*(Consensus) is strictly weaker than uniform Consensus$_{p.o.}$, *trans*(Ω) is especially necessary for uniform Consensus$_{p.o.}$. To show that *trans*(Ω) is sufficient for uniform Consensus$_{p.o.}$, we can simply use Theorem 7.11, since we know that Ω is sufficient for uniform Consensus with a majority of correct processes. □

Chapter 8

Byzantine Robust Protocols

Now we present our results about how to save on the number of processors when executing byzantine robust protocols, besides certain others, within a class of weaker and stronger assumptions. More precisely, here we initiate the first study of how to save on processors when running threshold protocols in a dependent failure model where processors may behave badly in an arbitrary, byzantine way [80]. In particular, we look at the problem of running any protocol that has a threshold upper bound of $n > kt$, for any positive integer constant k, where n is the total number of processors and t is the maximum number of failures. We introduce an *optimal byzantine-resilient transformation protocol* that enables any protocol with such a restriction to be automatically translated into a dependent failure model so that it can execute with less than n processors, saving computing power. No authentication, self-verification or encryption is needed.

To model dependent failures, we use the powerful notion of a *survivor set system*, the unique collection of all minimal sets of correct processors, each set containing all correct processors of some execution [74]. Survivor set systems are the mirrors of fail-prone systems [74], relating in a bijective complementary way, and can represent concisely complex adversarial structures [70], in a manner particularly useful for quorum systems. To further illustrate why such an abstraction is omnipotent, the existence of quorum systems [86] in such adversarial structures depends totally on the survivor set system. The reason is that, unlike quorum systems, survivor set systems encapsulate availability by definition, that is, for any execution, at least one set must have all its elements alive. Such availability requirement is crucial for almost all sensible quorum systems [86], thus implying that each element of the survivor set system must contain a quorum for a quorum system to exist. Moreover, as general adversarial structures, survivor set systems allow solutions to specific problems for which one would be impossible if a threshold model were to be used instead. Finally, we characterize *equivalence classes of adversarial structures*, in terms of solvability, by making use of a particular set of hierarchic properties based on set intersection.

In Section 8.1 we motivate work and in Section 8.2 we give details of the system model. Then in Section 8.3 we define essencial intersection properties to be used in the next Sections 8.4 and 8.5, presenting results and equivalence classes of adversarial structures.

8.1 Motivation

A typical way to model failures in a distributed system is to assume that no more than t out of n components can be faulty. Lower bounds for problems are often stated using such notation. For instance, it is widely known that consensus in a synchronous system with byzantine failures requires $n > 3t$ processors if digital signatures are not to be used [80]. That sort of failure representation is what is called *threshold model* in the literature: no more than t components out of n can fail in a distributed system, given a failure type. Hence, provided with previous definition, one can then design protocols that are optimal with respect to threshold-based lower bounds. Such protocols are optimal, however, only when failures are independent and identically distributed (IID). This is because the threshold assumption does not restrict which subsets of t or less processors can fail. To use a threshold algorithm in a system in which components can fail in an non-IID manner, one can compute t as being the largest number of faulty components in any execution. Then, if the number of components n is sufficiently large to meet the threshold-based requirements of the protocol based on this value of t, one can run the protocol on the system.

However, this approach can result in excessive replication. In [75], Junqueira and Marzullo developed a dependent failure representation that allows one to express bounds on problems concisely and in a specially practical way for quorum systems [86]. It is based on a *survivor set system*, the unique collection of all minimal sets of correct processors, each set containing all correct processors of some execution. Moreover, in [75], we also rederived some of the lower bounds for consensus using our dependent failure model and developed protocols that are optimal with respect to these bounds. More precisely, we showed that there are situations in which using the strategy described above—choosing t to be the largest number of faulty processors in any execution and having only one automaton running distributed protocol per processor—would fail because there would not be enough processors n to satisfy the replication bound $n > k \cdot t$. Nevertheless, there would in fact be enough if our dependent failure model was to be used.

Hence, we look closely at the problem of running a protocol that has a replication requirement of $n > k \cdot t$ for any positive integer constant k on systems in which failures are not IID. We develop an *optimal byzantine-resilient transformation protocol* that enables any protocol with such a restriction to be automatically translated into a dependent failure model so that it can execute with less than n processors, saving computing power. No authentication, self-verification or encryption is needed. Less general transformations for structural failure models regarding more particular problems were investigated by Warns, Freiling and Hasselbring in [115]. We also unveal *equivalence classes of adversarial structures*, in terms of solvability, by investigating a key set of hierarchic properties based on set intersection.

8.2 System model

A system is composed of a set of processors $\Pi = \{p_1, p_2, \cdots, p_n\}$ that communicate by exchanging messages. Each processor is capable of executing multiple automata, as in the model described by Attiya and Welch for simulations [9]. The set of automata of a system is $\{sm_1, sm_2, \cdots, sm_m\}$, where each automaton sm_i has a state, code, and an identity which is used by other automata to send messages to sm_i. The code of an automaton is deterministic, in that the state of an automaton is determined only from its initial state and the sequence of messages it has received. We model an automaton as a state machine [110].

We assume that processors can fail arbitrarily. A faulty processor can crash, modify the content of messages arbitrarily, omit to send or receive messages, and behave in malicious ways. It is important to observe that the failure of a processor implies the failure of all automata in it. Thus, if a processor is arbitrarily faulty, then all of the automata running in that processor can exhibit arbitrary behavior. The converse is also true: if a processor is correct, then all automata running in it are correct as well.

We do not assume a threshold on the number of processor failures. Instead, we characterize failure scenarios by providing the *survivor set systems*, the unique collection of all minimal sets of correct processors, each set containing all correct processors of some execution [74, 75]. Informally, elements of a survivor set system, called survivor sets, generalizes subsets of size $n - t$ under the threshold model, where n is the total number of processor and t is again a threshold on the number of processor failures. Let ϕ be an execution from the set of all possible executions Φ for the protocol, and $Correct(\phi)$ the set of automata which never fail in ϕ, that is, which always remain correct in ϕ. More formally, we define a survivor set as follows:

Definition 8.1. A subset $S \subseteq \Pi$ is a survivor set if and only if: 1) $\exists \phi \in \Phi$, $Correct(\phi) = S$; 2) $\forall \phi \in \Phi, p_i \in S$, $Correct(\phi) \not\subset S/p_i$.

Henceforth, we refer to the survivor set system as the collection of survivor sets of Π as S_Π. Note that it is unique given a fail-prone system [74], that is, an exact configuration of possible failures. The inverse is also true.

A survivor set system is equivalent to a core system [74], and one defines uniquely the other as well. Informally, a core is a subset of processors that generalizes subsets of size $t + 1$ in the threshold model, where t is a threshold on the number of processor failures. Thus, in every execution of the system, there is at least one processor in every core that is correct. From the set of cores, one can obtain the set of survivor sets by creating all minimal subsets of processors that intersect every core. More formally, we define cores as follows:

Definition 8.2. A subset $C \subseteq \Pi$ is a core if and only if: 1) $\forall \phi \in \Phi$, $Correct(\phi) \cap C \neq \emptyset$; 2) $\forall p_i \in C$, $\exists \phi \in \Phi$ such that $C/p_i \cap Correct(\phi) = \emptyset$.

Henceforth, we refer to the set of cores of Π as C_Π. Along with Π, both collections of sets constitute a *system profile*. We use the notation $\langle \Pi, S_\Pi, C_\Pi \rangle$ to refer to a system profile. Note that, in fact, the notation is so for later simplicity in usage, though either S_Π or C_Π would be enough to define a system profile, as one can uniquely be obtained from the other, just as the fail-prone system representing the adversarial structure. For a system $\langle \Pi, C_\Pi, S_\Pi \rangle$ we assume that there is no processor $p_i \in \Pi$ in every survivor set of S_Π, what indicates a processor that never fails. Were this the case, then the solution to many problems in distributed computing would be trivial.

We now present two examples to motivate the use of survivor set systems. The first example illustrates the advantage of our approach when the probability of failure across processors is not the same. In the second example, we look at a system in which failures of processors are partially correlated.

Example 8.3. :
Consider a system with five processors $\Pi = \{p_1, p_2, p_3, p_4, p_5\}$, and the following probabilities of failure:

$$\begin{aligned} P(p_1 \text{ is faulty in an execution}) &= P(p_2 \text{ is faulty in an execution}) = \\ &= P(p_3 \text{ is faulty in an execution}) = 0.01 \\ P(p_4 \text{ is faulty in an execution}) &= P(p_5 \text{ is faulty in an execution}) = 0.001 \end{aligned}$$

This means that p_1, p_2, and p_3 are not as reliable as p_4 and p_5. Assuming both that these processors fail independently, and that target degree of reliability for this system is 0.0001, we can then infer the following survivor set system and core system:

- $S_\Pi = \{\{p_1, p_4, p_5\}, \{p_2, p_4, p_5\}, \{p_3, p_4, p_5\}, \{p_1, p_2, p_3, p_4\}, \{p_1, p_2, p_3, p_5\}\}$.
- $C_\Pi = \{\{p_1, p_2, p_3\}, \{p_1, p_4\}, \{p_1, p_5\}, \{p_2, p_4\}, \{p_2, p_5\}, \{p_3, p_4\}, \{p_3, p_5\}, \{p_4, p_5\}\}$.

From [75], this system satisfies Byzantine Intersection and Byzantine Partition, two equivalent properties that are necessary and sufficient to solve consensus in a synchronous system with arbitrarily faulty processors. Consequently, one can solve this problem with seven automata but only five processors. Under the threshold model, it requires at least seven processors for seven automata.

Example 8.4. :
Suppose a system with six processors $\{p_1, p_2, p_3, p_4, p_5, p_6\}$, and the following properties for these processors:

- All the processors have the same probability x of failure in an execution;
- We can separate the processors in two distinct groups: $A = \{p_1,\ p_2,\ p_3\}$ and $B = \{p_4,\ p_5,\ p_6\}$;
- Let $\phi \in \Phi$: $P(p_i \in A \text{ is faulty in } \phi \mid p_j \in B \text{ is faulty in } \phi) = P(p_i \in A \text{ is faulty in } \phi) \cdot P(p_j \in B \text{ is faulty in } \phi)$(independent failures for processors in different groups);
- Let $\phi \in \Phi$: $P(p_i \in \Psi \text{ is faulty in } \phi \mid p_j \in \Psi \text{ is faulty in } \phi) > P(p_i \in \Psi \text{ is faulty in } \phi) \cdot P(p_j \in \Psi \text{ is faulty in } \phi)$, $i \neq j$ and $\Psi \in \{A, B\}$ (failures are positively correlated for processors in the same group) ;
- Let $\phi \in \Phi$: $P(p_i \in \Psi \text{ is faulty in } \phi \mid p_j, p_k \in \Psi \text{ are faulty in } \phi) < x^2$, $i \neq j, k$ and $\Psi \in \{A, B\}$;

Assuming that the target degree of reliability for this system is x^2, we can infer the survivor set system and core system:

- $S_\Pi = \{\{p_1, p_2, p_3, p_4\},\ \{p_1, p_2, p_3, p_5\},\ \{p_1, p_2, p_3, p_6\},\ \{p_1, p_4, p_5, p_6\},\ \{p_2, p_4, p_5, p_6\},\ \{p_3,\ p_4,\ p_5,\ p_6\}\}$.

- $C_\Pi = \{\{p_1, p_2, p_3\}, \{p_4, p_5, p_6\}, \{p_1, p_4\}, \{p_1, p_5\}, \{p_1, p_6\}, \{p_2, p_4\}, \{p_2, p_5\}, \{p_2, p_6\}, \{p_3, p_4\}, \{p_3, p_5\}, \{p_3, p_6\}\}$.

Suppose an implementation of a fault-tolerant state machine that tolerates arbitrary failures, such as the one described by Castro and Liskov [24]. To make failures independent, one can use opportunistic n-version programming, as proposed by Castro, Rodrigues and Liskov in [25]. If only two implementations are available, then one could analyze the failure behavior of these implementations. If it happens that the properties of these two implementations fulfill the ones described above, then such a system can be used to solve the given problem, since one can be used in group A and the other in group B. Note that if there is a high probability that all processors executing the same implementation fail together, then this construction is not useful.

It is important to observe that our dependent failure model does not provide an approach that violates impossibility results previously presented, such as the one on the minimum degree of replication necessary to solve consensus with arbitrary failures [80]. We observe, however, that under realistic assumptions, our approach is able, in several instances, to overcome impossibility results.

8.3 Replication with cores and survivor sets

In [75], Junqueira and Marzullo showed two equivalent properties on replication that are necessary and sufficient to solve consensus assuming arbitrary processor failures under the core and survivor set systems model. These properties, called *Byzantine Partition* and *Byzantine Intersection*, generalize the bound on replication based on a threshold $n > 3t$, where n is the number of processors in a system and t is a threshold on the number of processors failures. Based on these properties, we stated two parameterized properties (α, β)-Partition and (α, β)-Intersection, for integers α, β and $\alpha > \beta \geq 1$, that generalizes a bound of the form $n > \lfloor \alpha t / \beta \rfloor$. An example of such a bound in the literature is the lower bound for primary-backup with receive-omission failures, which is $n > \lfloor 3t/2 \rfloor$ [23].

In this work, we concentrate on the cases in which $\beta = 1$ and $\alpha = k \geq 2$, which is equivalent in the threshold model to a replication requirement of $n > k \cdot t$ for $k \geq 2$. This replication bound implies that if one constructs k subsets of the processors, then at least one of them will contain at least $n - t$ processors. Generalized to an expression on cores, we have:

Property 8.5. : ***k*-Partition**
For every partition $\mathcal{A} = \{A_1, A_2, \cdots, A_k\}$ of Π, at least one of the sets A_i contains a core. $\Box$

For the following sections, we make use of the equivalent $(k, 1)$-Intersection property, which we call from this point on k-Intersection. This is a property expressed in terms of survivor sets rather than cores, and it is essencial both for our optimal automatic translation protocol and the definition of equivalence classes of adversarial structures. Let $\langle \Pi, C_\Pi, S_\Pi \rangle$ be a system profile such that there is no core in C_Π of size one. Then:

Property 8.6. : ***k*-Intersection**
For every $\{S_1, S_2, \cdots, S_k\} \in S_\Pi$, $\cap_i S_i \neq \emptyset$. $\Box$

8.4 Constructing protocols

In this section, we present ways to build protocols for our dependent failure model out of protocols designed for the threshold model. By assumption, protocol Λ_t requires $n > k \cdot t$ replication for some positive integer value of k and some threshold on the number of failures $t > 0$. The main idea is to allow a processor to run more than one automaton of Λ_t. At first glance, this may appear to be a fruitless approach, since automata being executed by the same processor fail in a completely correlated fashion. That is, a processor p_i being faulty is equivalent to all of the automata of Λ_t that p_i executes being faulty as well. By choosing which automata to replicate, however, one can increase replication enough without increasing t in a similar manner so that the replication requirement $n > k \cdot t$ is met.

Our goal is to provide a method for constructing a protocol Λ_{CS} for the core and survivor set systems model based on protocol Λ_t. More specifically, given a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$ satisfying k-Intersection, we provide a set $\mathcal{M} \langle \Lambda_{CS} \rangle$ of automata and a mapping $\varphi \langle \Lambda_{CS} \rangle$ of these automata to processors.

First, we describe a procedure to determine a value of n and to assign automata of Λ_t to processors, such that in no execution more that t automata of Λ_t fail, for some value of t under the constraint that $n > k \cdot t$. Consider a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$. Let l_i be the fraction of automata of $\mathcal{M}(\Lambda_t)$ that processor $p_i \in \Pi$ executes, where $0 \leq l_i < 1$. Moreover, in each execution, the fraction of correct automata is as follows:

$$\frac{n-t}{n} > \frac{(k \cdot t - t)}{k \cdot t} = \frac{k-1}{k}$$

These observations lead us to the following set of constraints:

$$\begin{aligned} \sum_{p_i \in \Pi} l_i &= 1 \\ \forall s \in S_\Pi : \sum_{p_i \in s} l_i &> \frac{k-1}{k} \end{aligned} \tag{8.1}$$

These equations imply that every processor is executed by exactly one processor, and in no execution there are more than $\lfloor n/k \rfloor$ faulty automata. If we solve this system of linear equations, and choose a large enough value of n such that $n \cdot l_i$ is an integer for every i, then we have a solution for our problem. We can then simply choose the smallest value of n for which this condition holds. That is:

$$\min_n \{\forall l_i : n \cdot l_i \text{ is an integer}\} \tag{8.2}$$

Let $\langle \Pi, C_\Pi, S_\Pi \rangle$ be a system profile and $\mathcal{L}$ be a set of values l_i, $0 \leq l_i < 1$, with a value l_i for each processor of Π. We say that $\mathcal{L}$ is a valid solution for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and degree of replication k if these values satisfy the constraints in Definition 8.1 for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and k. The following theorem states that there being a valid solution is sufficient for a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$ to satisfy k-Intersection.

Theorem 8.7. *Let $\langle \Pi, C_\Pi, S_\Pi \rangle$ be a system profile and k be a degree of replication. There is a valid solution $\mathcal{L}$ for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and k only if $\langle \Pi, C_\Pi, S_\Pi \rangle$ satisfies k-Intersection.*

Proof. We prove this theorem by contradiction. Suppose that there is a system with profile $\langle \Pi, C_\Pi, \mathcal{S}_\Pi \rangle$ that does not satisfy k-Intersection and there is a valid array of values l_i for $\langle \Pi, C_\Pi, \mathcal{S}_\Pi \rangle$ and k. From the first assumption, there is a subset $\{S_1, S_2, \cdots, S_k\}$ in $\mathcal{S}_\Pi$ such that $\cap_i S_i = \emptyset$. From the second assumption we have the following:

$$\begin{aligned}\sum_{p_i \in S_1} l_i &> \frac{k-1}{k} \\ \sum_{p_i \in S_2} l_i &> \frac{k-1}{k} \\ &\vdots \\ \sum_{p_i \in S_k} l_i &> \frac{k-1}{k}\end{aligned}$$

Summing together these k equations, we have the following:

$$\left(\sum_{p_i \in S_1} l_i\right) + \left(\sum_{p_i \in S_2} l_i\right) + \cdots + \left(\sum_{p_i \in S_k} l_i\right) > (k-1) \tag{8.3}$$

Since by assumption no processor is in all k survivor sets by assumption and $k \geq 2$ we also have that:

$$\left(\sum_{p_i \in S_1} l_i\right) + \left(\sum_{p_i \in S_2} l_i\right) + \cdots + \left(\sum_{p_i \in S_k} l_i\right) \leq \sum_{p_i \in \Pi} l_i \leq 1 \leq (k-1) \tag{8.4}$$

Equations 8.3 and 8.4, however, contradict each other, giving us our contradiction. □

Using the value of n provided by (8.2), we can determine the set of automata $\mathcal{M}\langle \Lambda_t \rangle$. Assuming a valid solution $\mathcal{L}$, we build $\mathcal{M}\langle \Lambda_{CS} \rangle$ and $\varphi\langle \Lambda_{CS} \rangle$ as follows:

1. $\mathcal{M}\langle \Lambda_{CS} \rangle \leftarrow \mathcal{M}\langle \Lambda_t \rangle$;
2. For each $sm \in \mathcal{M}\langle \Lambda_{CS} \rangle$: $\varphi\langle \Lambda_{CS} \rangle \leftarrow [\varphi\langle \Lambda_{CS} \rangle \mid sm_i \rightarrow p_j]$, $(p_j \in \Pi) \wedge (|(\varphi\langle \Lambda_{CS} \rangle)^{-1}(p_j)| \leq n \cdot l_j)$.

The following theorem states that in every execution of Λ_{CS} there are at most t faulty automata, where $t = n - \lfloor \frac{k \cdot (n+1) - n}{k} \rfloor$.

Theorem 8.8. *Let $\langle \Pi, C_\Pi, \mathcal{S}_\Pi \rangle$ be a system profile and k a degree of replication. Given a valid solution $\mathcal{L}$ for $\langle \Pi, C_\Pi, \mathcal{S}_\Pi \rangle$ and k, there are at most t faulty automata in any execution $\phi \in \Phi$ of the protocol Λ_{CS}, for $t = n - \lfloor \frac{k \cdot (n+1) - n}{k} \rfloor$.*

Proof. We first show that in every execution, there are at least $n - t$ correct automata, for some value of t. By assumption, for every execution $\phi \in \Phi$ there is at least one survivor set $S \in \mathcal{S}_\Pi$ containing only correct processors. Given that $\mathcal{L}$ is a valid solution, we have:

$$\sum_S n \cdot l_i = n \cdot \sum_S l_i > \frac{n \cdot (k-1)}{k} \geq \left\lfloor \frac{n \cdot (k-1)}{k} + 1 \right\rfloor \tag{8.5}$$

It also must be the case that for all $S \in S_\Pi$, the sum of t and $\sum_S n \cdot l_i =$ is greater or equal to n. Otherwise, there is at least one execution in which there are more than t faulty automata. Expressed more formally,

$$\begin{aligned} \forall S \in S_\Pi : \quad & n \cdot \sum_S l_i + t \geq n \\ \Rightarrow \quad & \forall S \in S_\Pi : t \geq n - n \cdot \sum_S l_i && (8.6) \\ \Rightarrow \quad & t \geq n - \min_{S \in S_\Pi} \{ n \cdot \sum_S l_i \} && (8.7) \end{aligned}$$

By equation 8.5, we have that the value of the previous sum is bounded from below by $\left\lfloor \frac{k \cdot (n+1) - n}{k} \right\rfloor$. That is:

$$\min_S \{ n \cdot \sum_S l_i \} \geq \left\lfloor \frac{k \cdot (n+1) - n}{k} \right\rfloor \tag{8.8}$$

If we then choose t as:

$$t = n - \left\lfloor \frac{k \cdot (n+1) - n}{k} \right\rfloor \tag{8.9}$$

We have:

$$\forall S \in S_\Pi : n - n \cdot \sum_S l_i \leq n - \left\lfloor \frac{k \cdot (n+1) - n}{k} \right\rfloor = t \tag{8.10}$$

From Equation 8.10, we conclude that there is no execution in which more that t automata fail, where t is given by Equation 8.9.

□

With this construction, automata behave as in the original protocol, sending and receiving messages from each other. The only difference is that some automata may run in the same processor, and consequently these automata fail together. From the previous theorem, however, our construction provides a threshold on the number of faulty automata that does not violate the replication requirement for protocol Λ_t, thereby guaranteeing the correct execution of Λ_{CS}.

A problem is that, in some instances, even if a system profile satisfies k-Intersection, for some value of k, there is no valid solution for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and k. We show this with the following theorem, and explain in the sequence how to circumvent optimally this impossibility result.

Theorem 8.9. *For every value of $k > 1$, there is a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$ satisfying k-Intersection such that there is no valid solution $\mathcal{L}$ for $\langle \Pi, C_\Pi, S_\Pi \rangle$ and k.*

Proof. The case $k = 2$ follows directly from Theorem 3.1 in [57]. We show that it holds for $k \geq 3$.

We construct a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$ for which the proposition holds. Suppose that $|\Pi| = (k-1) \cdot k$. Now, partition Π into $k-1$ disjoint sets $A = (A_1, A_2, \cdots, A_{k-1})$, each of size k, and let C_Π be as follows:

$$C_\Pi = \{A_1, A_2, \cdots, A_{k-1}\} \cup \{\{p_i, p_j\} \mid p_i \in A_x, p_j \in A_y, x \neq y\} \tag{8.11}$$

From this set of cores, we can build the set of survivor sets as follows:

$$S_\Pi = \{A_x \cup \{p_i\} \mid p_i \in A_y, x \neq y\} \tag{8.12}$$

This system clearly satisfies k-partition, since any partition of Π into k subsets will result in at least one subset containing either all of some A_x or two automata from different subsets A_x and A_y. We now show that there is no set of values l_i, one for each processor $p_i \in \Pi$, satisfying equations in 8.1.

The set of linear equations for our system is as follows. For each A_x and $p \in A_x$:

$$\left(\sum_{A_y \in A/\{A_x\}} \sum_{p_i \in A_y} l_i \right) + l_p > \frac{k-1}{k} \tag{8.13}$$

From 8.13, there are $k \cdot (k-1)$ equations, where each l_i appears on the left side of exactly $k \cdot (k-2) + 1 = (k-1)^2$ equations. Summing up each side these equations, we get:

$$\begin{aligned}
& (k-1)^2 \cdot (l_1 + l_2 + \cdots + l_{|\Pi|}) > k \cdot (k-1) \cdot \frac{k-1}{k} \\
\Rightarrow \quad & (k-1)^2 \cdot (l_1 + l_2 + \cdots + l_{|\Pi|}) > (k-1)^2 \\
\Rightarrow \quad & (l_1 + l_2 + \cdots + l_{|\Pi|}) > 1 && (8.14) \\
& && (8.15)
\end{aligned}$$

We conclude that the constraint imposed by the first equation of 8.1 cannot be fulfilled as we wanted to show. □

To illustrate this construction, consider the system profile of example 8.3. Recall that in that system there are five processors. There is one core that contains three processors, and all of other cores are of size two. Given $k = 3$, we have the following solution for this system:

- $l_1 = \frac{1}{7}$, $l_2 = \frac{1}{7}$, $l_3 = \frac{1}{7}$, $l_4 = \frac{2}{7}$, $l_5 = \frac{2}{7}$;
- We choose $n = 7$.
- Let $\mathcal{M}\langle\Lambda_t\rangle = \{sm_1, sm_2, sm_3, sm_4, sm_5, sm_6, sm_7\}$. A possible mapping $\varphi\langle\Lambda_{cs}\rangle$ is as follows: $\{sm_1 \rightarrow p_1, sm_2 \rightarrow p_2, sm_3 \rightarrow p_3, sm_4 \rightarrow p_4, sm_5 \rightarrow p_4, sm_6 \rightarrow p_5, sm_7 \rightarrow p_5\}$;
- $t = n - \lfloor \frac{k\cdot(n+1)-n}{k} \rfloor = 7 - 5 = 2$;

In example 8.4, however, there is no solution satisfying the set of constraints 8.1. This example is actually the case presented in the proof of Theorem 8.4 for $k = 3$.

However, remark that, to circumvent the impossibility result, it suffices to apply Byzantine Consensus[75, 21] in an optimal fashion. Note that resorting to Byzantine Consensus is actually necessary, once having state machine replication [110, 24] in a timely manner is required.

8.5 Equivalence classes of adversarial structures

To end, we would like to point out that the hierarchy of k-intersection properties defines then in a straightforward fashion, due to previous section, equivalence classes of adversarial structures in terms of problem solvabilities:

Theorem 8.10. *Any protocol requiring $n > kt$ replication is solvable for a system profile $\langle \Pi, C_\Pi, S_\Pi \rangle$ with $|\Pi| \leq n$ if and only if $\langle \Pi, C_\Pi, S_\Pi \rangle$ satisfies k-Intersection.*

Chapter 9

Summary and Future Work

In this thesis we presented several techniques to protect against hostile attacks in a message-passing distributed computing environment. We started by considering relatively benign failures and proceeded to more malign ones. We focused on the classical k-set agreement problem. We derived an optimal crash-resilient k-set agreement protocol, which tolerates an optimal linear number of crashes given (exactly or close to) minimal synchrony features provided by limited-scope failure detectors in asynchronous systems. Tight bounds on the maximum possible number of crashes were achieved through techniques borrowed from combinatorial topology. How to connect failure detection to timing assumptions was also discussed.

Next, we considered message omission failure models in the form of a tamper-proof secure coprocessor embedded in an untrusted host. The coprocessor exchanged messages with other secure coprocessors at other untrusted hosts. The untrusted host could fail to send or deliver messages, but could not tamper with them. Using secret shared coins, we developed randomized consensus (that is, 1-set agreement) protocols, optimal both in terms of time and resilience, with applications to secure electronic commerce. Deterministic versions and automatic transformations for failure detectors were also given.

Finally, we turned our attention into Byzantine faults. We introduced an optimal byzantine-resilient protocol that allows one protocol using fewer processors to simulate another protocol that uses more. We also identified a particular hierarchy of properties that define equivalence classes of failure configurations.

As an open problem in Chapter 3, we point out that it is still unknown which is the weakest failure detector for the problem of k-set agreement. An initial investigation towards that goal was made in [92], where a failure detector Ω_k, generalizing Ω, was compared to others and presented as a potential candidate.

In chapter 4, it is not known whether and how results would map to more severe failures. Still another research direction, stemming from Chapter 5, would be one concerning resilience improvement in randomized consensus protocol applied in secure eletronic commerce, more specifically one which requires a majority of correct hosts. For that, techniques from quantum cryptography could be useful in turning the protocol wait-free, that is, into a state where execution of any coprocessor would not depend on the correct execution of any other coprocessor. In that case, instead of a majority of correct processes, only one correct process would be necessary for a secure eletronic commerce protocol to take place, which would mean a huge gain. Moreover, note that both the ConsensusS and ConsensusSR binary consensus protocols can be extended to a larger set of k values in $3\log(k)$ rounds via bit-by-bit consensus. It is an open question whether faster protocols exist (perhaps by doing bit-by-bit consensus in parallel).

Note that in Chapter 6 there are multiple lines of future work to consider. On the practical side as next step we intend to have the approach implemented and extended to other classes of failure detectors. On the theoretical side it would be interesting to study the minimal storage and communication effort necessary to solve consensus, since we use unbounded buffers in our implementation and the bit complexity of the messages we use is rather high. Also it is necessary to investigate the timing assumptions still further since in theory smartcards can also be slowed down arbitrarily. In such cases the assumptions of partial synchrony may not hold and we come close to a truly asynchronous system where consensus cannot be solved [52]. Investigating realistic models of smartcard-based systems that reflect this type of attack and still allow TrustedPals to be implemented will further broaden the applicability of the framework in practice.

In the trusted system however, security modules may not have their own source of activation (and therefore no realtime clock), so an untrusted host *can* on the one hand slow down its smartcard arbitrarily by disturbing the clock speed. On the other hand, the host *cannot* speed up the clock of the security module arbitrarily. This still results in a model of partial synchrony for the trusted system as we now explain.

Consider for example the situation depicted in Fig. 9.1. There a channel directed from a security module q to a security module p is shown. The correct process q sends heartbeat messages in constant time intervals to the malicious process p. The usual strategy in this case is that the receiver p sets a timeout $\Delta_p(q)$ of p for q. If p does not receive a message from q within this timeout interval, p suspects q to have failed. Now, if p manipulates its clock so that its smartcard processes faster and as a result the timeout intervals $\Delta_p(q)$ are shortened, p will assume q to have failed until it receives a message from q. It might happen finitely often that p times-out on q. After each timeout on q $\Delta_p(q)$ grows, and eventually the bounds on process speed and message delay hold. Thus, eventually process p cannot timeout on process q anymore, and the malicious host will have no impact on the system's timing assumption.

The inverse scenario is shown in Fig. 9.2. Here, the channel from a malicious host q to a correct host p is considered. Process q may intentionally withhold a message m and delay the sending of m for an arbitrarily amount of time. This will cause process p to assume that q has failed. If q finally sends m, p will not assume q to have failed anymore. This might also happen infinitely often. Assuming the system to be synchronous or partially synchronous and that there are no send omissions, the arbitrary delay caused by the malicious process will violate the timing assumption of the system.

In Chapter 6, as an open problem, we think that it would be interesting to replace the requirement of a correct majority in our second transformation with a failure detector Σ [39] that will also be sufficient. Apart from that, it may be possible to give more specific transformations that are less general, but also less communication expensive than our transformation.

In special, we would like to expand work in Chapter 7 on protocol simulation with fewer processors in adversarial structure models. First of all, it would be useful to have as well simulations for protocols with fractional thresholds instead of only integer ones. Such thresholds are particularly important to weak leader election [23], k-mutex [76] and fast paxos [88]. Second, it would be equally interesting to introduce probabilities on the existence of elements belonging to adversarial structures. Third, a very appealing scenario which recently has been gaining attention is a hybrid one where processors may be guided to maximize distinct specific utility functions, instead of the same one, as it is assumed in secure electronic commerce. Further investigation on the exact meaning of higher level hierarchy properties and on how precisely such properties relate to existing literature could

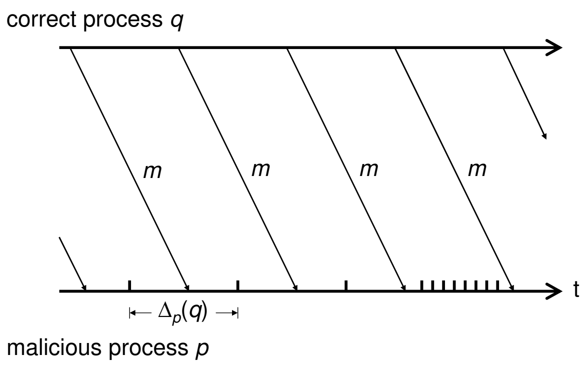

Figure 9.1: Channel directed from a correct process q to a malicious process p.

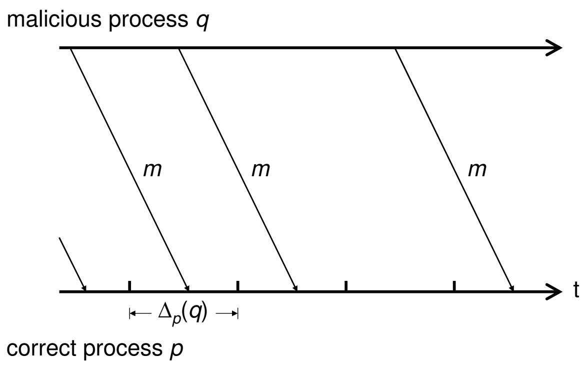

Figure 9.2: Channel directed from a malicious process q to a correct process p.

clarify many open questions, including concerning paxos functioning and optimal efficiency despite different circumstances. Hence, other types of optimality could be achieved as well with protocol simulation depending on the assumptions present in the given adversary structure model.

Last, we would join a recent trend [119], and would thrive to develop in the near future fast heuristics to find most efficient quoruns given a survivor set system (or equivalently, a fail-prone system) and a minimal set of properties required to solve a problem. Such heuristics could, for instance, be inspired from existing ones in combinatorial optimization such as local search, simulated annealing, branch and bound, or a combination of any of those or other techniques, nowadays also useful in scheduling american and european sport competitions.

Bibliography

[1] B. Charron-Bost A. Basu and S. Toueg. Revisiting safety and liveness in the context of failures. In *Proceedings of the 11th International Conference on Concurrency Theory*, pages 552–565. Springer-Verlag, 2000.

[2] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Stable leader election. In *Proceedings of the 15th International Conference on Distributed Computing*, pages 108–122. Springer-Verlag, 2001.

[3] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. On implementing Omega with weak reliability and synchrony assumptions. In *Proceeding of the 22nd Annual ACM Symposium on Principles of Distributed Computing.* ACM Press, 2003.

[4] M. K. Aguilera, C. Delporte-Gallet, H. Fauconnier, and S. Toueg. Communication-efficient leader election and consensus with limited link synchrony. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 328–337. ACM Press, 2004.

[5] B. Altmann, M. Fitzi, and U. Maurer. Byzantine agreement secure against general adversaries in the dual failure model. In *Proceedings of the Thirteenth DISC*, pages 123–139, September 1999.

[6] E. Anceaume, M. Hurfin, and P. R. Parvedy. An efficient solution to the k-set agreement problem. Technical Report PI-1440, INRIA, July 2003.

[7] J. Aspnes. Randomized protocols for asynchronous consensus. *Distributed Computing*, 16(2–3):165–175, September 2003.

[8] H. Attiya and Z. Avidor. Wait-free n-set consensus when inputs are restricted. In D. Malkhi, editor, *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings*, volume 2508 of *Lecture Notes in Computer Science*, pages 326–338. Springer, 2002.

[9] H. Attiya and J. Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill, 1998.

[10] H. Attiya and J. Welch. *Distributed Computing.* John Wiley & Sons, 2nd edition, 2004.

[11] G. Avoine, F. Gärtner, R. Guerraoui, and M. Vukolic. Gracefully degrading fair exchange with security modules. In *Proceedings of the Fifth European Dependable Computing Conference*, pages 55–71. Springer-Verlag, April 2005.

[12] G. Avoine and S. Vaudenay. Optimal fair exchange with guardian angels. In *International Workshop on Information Security Applications (WISA), LNCS*, volume 4, 2003.

[13] O. Babaoglu, R. Davoli, and A. Montresor. Group communication in partitionable systems: Specification and algorithms. *IEEE Trans. Softw. Eng.*, 27(4):308–336, 2001.

[14] A. Basu, B. Charron-Bost, and S. Toueg. Simulating reliable links with unreliable links in the presence of process crashes. In *Proceedings in the 10th International Workshop on Distributed Algorithms (WDAG96)*, pages 105–122, 1996.

[15] R. A. Bazzi and G. Neiger. Simulating crash failures with many faulty processors (extended abstract). In *WDAG '92: Proceedings of the 6th International Workshop on Distributed Algorithms*, pages 166–184, London, UK, 1992. Springer-Verlag.

[16] Z. Benenson, F. Freiling, T. Holz, D. Kesdogan, and L. D. Penso. Safety, liveness, and information flow: Dependability revisited. In *Proceedings of the 4th ARCS International Workshop on Information Security Applications*, pages 56–65. Springer-Verlag, March 2006.

[17] Z. Benenson, F. C. Gärtner, and D. Kesdogan. Secure multi-party computation with security modules. Technical Report AIB-10-2004, RWTH Aachen, December 2004.

[18] M. Biely, M. Hutle, L. D. Penso, and J. Widder. Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency. In *Proceedings of the Ninth International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 4–20. Springer-Verlag, November 2007.

[19] M. Biely, M. Hutle, L. D. Penso, and J. Widder. Relating stabilizing timing assumptions to stabilizing failure detectors regarding solvability and efficiency. Technical Report 54/2007, Technische Universität Wien, Institut für Technische Informatik, Treitlstrasse 3, 2nd floor, 1040 Wien, E.U., August 2007.

[20] M. Biely and J. Widder. 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)*, volume 4280 of *LNCS*, pages 110–121. Springer Verlag, November 2006.

[21] E. Borowsky and E. Gafni. Generalized FLP impossibility result for t-resilient asynchronous computations. In *Proceedings of the Twenty-Fifth ACM Symposium on Theory of Computing*, pages 91–100. ACM Press, May 1993.

[22] J. Bowen and V. Stravridou. Safety-critical systems, formal methods and standards. *IEE/BCS Software Engineering Journal*, 8(4):189–209, July 1993.

[23] N. Budhiraja, K. Marzullo, F. Schneider, and S. Toueg. Optimal primary-backup protocols. In *Proceedings of the Sixth WDAG*, pages 362–378, November 1992.

[24] M. Castro and B. Liskov. Practical byzantine fault-tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 20:398–461, November 2002.

[25] M. Castro, R. Rodrigues, and B. Liskov. Base: Using abstraction to improve fault tolerance. *ACM Transactions on Computer Systems*, 21:236–269, August 2003.

[26] J. Chalopin, S. Das, and N. Santoro. Rendezvous of mobile agents in unknown graphs with faulty links. In *Proceedings of the Twenty-First International Symposium on Distributed Computing, (DISC)*, pages 24–26. Springer-Verlag, September 2007.

[27] T. D. Chandra, V. Hadzilacos, and S. Toueg. The weakest failure detector for solving consensus. *Journal of the ACM (JACM)*, 43(4):685–722, 1996.

[28] T. D. Chandra and S. Toueg. Unreliable failure detectors for reliable distributed systems. *Journal of the ACM*, 43(2):225–267, March 1996.

[29] B. Charron-Bost, R. Guerraoui, and A. Schiper. Synchronous system and perfect failure detector: solveability and efficiency issues. In *Proceedings of the International Conference on Dependable System and Networks*. IEEE Computer Society Press, 2000.

[30] B. Charron-Bost and A. Schiper. The heard-of model: Unifying all benign failures. Technical Report LSR-REPORT-2006-004, EPFL, 2006.

[31] S. Chaudhuri. Agreement is harder than consensus: set consensus problems in totally asynchronous systems. In *Proceedings of the Ninth Annual ACM Symosium on Principles of Distributed Computing*, pages 311–234, August 1990.

[32] S. Chaudhuri, M. Herlihy, N. A. Lynch, and M. R. Tuttle. Tight bounds for k-set agreement. *Journal of the ACM (JACM)*, 47(5):912–943, 2000.

[33] Z. Chen. *Java Card Technology for Smart Cards - 1st Edition*. Addison-Wesley Professional, 2000.

[34] R. Cleve. Limits on the security of coin flips when half the processors are faulty (extended abstract). In *Proceedings of the Eighteenth Annual ACM Symposium on Theory of Computing*, pages 364–369, Berkeley, California, 28–30 May 1986.

[35] M. Correia, P. Veríssimo, and N. F. Neves. The design of a COTS-Real-time distributed security kernel. In Fabrizio Grandoni and Pascale Thévenod-Fosse, editors, *Dependable Computing - EDCC-4, 4th European Dependable Computing Conference, Toulouse, France, October 23-25, 2002, Proceedings*, volume 2485 of *Lecture Notes in Computer Science*, pages 234–252. Springer, 2002.

[36] R. Cortiñas, F. Freiling, M. Ghajar-Azadanlou, A. Lafuente, M. Larrea, L. D. Penso, and I. Soraluze. Secure failure detection in trustedpals. In *Proceedings of the Ninth International Symposium on Stabilization, Safety, and Security of Distributed Systems*, pages 173–188. Springer-Verlag, November 2007.

[37] C. Delporte-Gallet, H. Fauconnier, F. Freiling, L. D. Penso, and A. Tielmann. Automatic transformations from crash-stop to permanent omission. HAL archives ouvertes, hal-00160626, 2007.

[38] C. Delporte-Gallet, H. Fauconnier, and F. C. Freiling. Revisiting failure detection and consensus in omission failure environments. In *ICTAC*, pages 394–408, 2005.

[39] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, V. Hadzilacos, P. Kouznetsov, and S. Toueg. The weakest failure detectors to solve certain fundamental problems in distributed computing. In *PODC '04: Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*, pages 338–346, New York, NY, USA, 2004. ACM Press.

[40] C. Delporte-Gallet, H. Fauconnier, R. Guerraoui, and B. Pochon. The perfectly-synchronised round-based model of distributed computing. *Information & Computation*, 205(5):783–815, May 2007.

[41] C. Deporte-Gallet, H. Fauconnier, F. Freiling, L. D. Penso, and A. Tielmann. From crash-stop to permanent omission: Automatic transformation and weak failure detectors. In *Proceedings of the Twenty-First International Symposium on Distributed Computing*, pages 165–178. Springer-Verlag, September 2007.

[42] D. Dolev, C. Dwork, and L. Stockmeyer. On the minimal synchronism needed for distributed consensus. *Journal of the ACM*, 34:77–97, 1987.

[43] D. Dolev, R. Friedman, I. Keidar, and D. Malkhi. Brief announcement: Failure detectors in omission failure environments. In *Proceedings of the 16th Annual ACM Symposium on Principles of Distributed Computing*, page 286. Springer-Verlag, July 1997.

[44] A. Doudou, B. Garbinato, and R. Guerraoui. Encapsulating failure detection: from crash to Byzantine failures. In *Proceedings of the Int. Conference on Reliable Software Technologies*, Vienna, May 2002.

[45] A. Doudou, B. Garbinato, R. Guerraoui, and A. Schiper. Muteness failure detectors: Specification and implementation. In Jan Hlavicka, Erik Maehle, and András Pataricza, editors, *EDCC*, volume 1667 of *Lecture Notes in Computer Science*, pages 71–87. Springer, 1999.

[46] P. Dutta, R. Guerraoui, and I. Keidar. The overhead of consensus failure recovery. *Distributed Computing*, 19(5–6):373–386, April 2007.

[47] P. Dutta, R. Guerraoui, and L. Lamport. How fast can eventual synchrony lead to consensus? In *Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 22–27, 2005.

[48] C. Dwork, N. A. Lynch, and L. Stockmeyer. Consensus in the presence of partial synchrony. *Journal of the ACM*, 35(2):288–323, April 1988.

[49] J. G. Dyer, M. Lindemann, R. Perez, R. Sailer, L. van Doorn, S. W. Smith, and S. Weingart. Building the IBM 4758 secure coprocessor. *IEEE Computer*, 34(10):57–66, October 2001.

[50] P. Feldman and S. Micali. Optimal algorithms for byzantine agreement. In *Proceedings of the Twentieth Annual ACM Symposium on Theory of Computing*, pages 148–161. ACM Press, May 1988.

[51] C. Fetzer, U. Schmid, and M. Süßkraut. 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)*, pages 271–280. IEEE Computer Society, June 2005.

[52] M. Fischer, N. Lynch, and M. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM (JACM)*, 32(2):374–382, April 1985.

[53] M. Fort, F. Freiling, L. D. Penso, Z. Benenson, and D. Kesdogan. Trustedpals: Secure multiparty computation implemented with smartcards. In *11th European Symposium on Research in Computer Security (ESORICS)*, pages 306–314. Springer-Verlag, September 2006.

[54] F. Freiling, M. Herlihy, and L. D. Penso. Optimal randomized fair exchange with secret shared coins. In *Proceedings of the Ninth International Conference on Principles of Distributed Systems*, pages 61–72. Springer-Verlag, December 2005.

[55] F. C. Freiling, R. Guerraoui, and P. Kouznetsov. The failure detector abstraction. Technical report, Department for Mathematics and Computer Science, University of Mannheim, 2006.

[56] E. Gafni. Round-by-round fault detectors (extended abstract): unifying synchrony and asynchrony. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 143–152. ACM Press, 1998.

[57] H. Garcia-Molina and D. Barbara. How to assign votes in a distributed system. *Journal of the ACM*, 32(4):841–860, October 1985.

[58] F. C. Gärtner and S. Pleisch. Failure detection sequencers: Necessary and sufficient information about failures to solve predicate detection. In *Proceedings of the 16th International Conference on Distributed Computing (DISC'02)*, pages 280–294, 2002.

[59] F. Greve and S. Tixeuil. Knowledge connectivity vs. synchrony requirements for fault-tolerant agreement in unknown networks. In *Dependable Systems and Networks (DSN 2007)*, pages 82–91, 2007.

[60] R. Guerraoui and L. Rodrigues. *Introduction to Reliable Distributed Programming.* Springer-Verlag, April 2006.

[61] V. Hadzilacos. *Issues of fault tolerance in concurrent computations (databases, reliability, transactions, agreement protocols, distributed computing).* PhD thesis, Harvard University, 1985.

[62] W. L. Heimerdinger and C. B. Weinstock. A conceptual framework for system fault tolerance. Technical Report CMU/SEI-92-TR-33, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, October 1992.

[63] M. Herlihy and J.Tygar. How to make replicated data secure. In *Proceedings of the 7th Annual International Cryptology Conference (CRYPTO 1987)*, pages 379–391, August 1987.

[64] M. Herlihy and L. D. Penso. Tight bounds for k-set agreement with limited-scope failure detectors. *Distributed Computing*, 18(2):157–166, July 2005.

[65] M. Herlihy and S. Rajsbaum. Algebraic spans. *Mathematical Structures in Computer Science*, 10(4):549–573, August 2000. Special Issue: Geometry and Concurrency.

[66] M. Herlihy, S. Rajsbaum, and M. R. Tuttle. Unifying synchronous and asynchronous message-passing models. In *Proceedings of the seventeenth annual ACM symposium on Principles of distributed computing*, pages 133–142. ACM Press, 1998.

[67] M. Herlihy, S. Rajsbaum, and M. R. Tuttle. A new synchronous lower bound for set agreement. In *Proceedings of DISC 2001*, pages 136–150, 2001.

[68] M. Herlihy and N. Shavit. The topological structure of asynchronous computability. *Journal of the ACM (JACM)*, 46(6):858–923, November 1999.

[69] M. P. Herlihy and J. M. Wing. Specifying graceful degradation. *IEEE Transactions on Parallel and Distributed Systems*, 2(1):93–104, January 1991.

[70] M. Hirt and U. Maurer. Complete characterization of adversaries tolerable in secure multi-party computation. In *Proceedings of the Sixteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 25–34, August 1997.

[71] M. Hutle, D. Malkhi, U. Schmid, and L. Zhou. Brief announcement: Chasing the weakest system model for implementing Ω and consensus. In *Proceedings of the 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems*, 2006.

[72] M. Hutle and A. Schiper. Communication predicates: A high-level abstraction for coping with transient and dynamic faults. In *Dependable Systems and Networks (DSN 2007)*, pages 92–101, 2007.

[73] F. Junqueira. *Coping with Dependent Failures in Distributed Systems.* Number 0737 in CS2003. Ph.D. Thesis, UC San Diego, September 2002.

[74] F. Junqueira and K. Marzullo. Designing algorithms for dependent process failures. *Future Directions in Distributed Computing (FuDiCo)*, 2584:24–28, January 2003.

[75] F. Junqueira and K. Marzullo. Synchronous consensus for dependent process failures. In *Proceedings of the Conference on Distributed Computing Systems (ICDCS)*, pages 274–283. Springer-Verlag, May 2003.

[76] H. Kakugawa, S. Fujita, M. Yamashita, and T. Ae. Availability of k-coterie. *ACM Transactions on Computers*, 42(5):553–558, 1993.

[77] G. Karjoth. Secure mobile agent-based merchant brokering in distributed marketplaces. In *Proceedings of the Second International Symposium on Agent Systems and Applications and Fourth International Symposium on Mobile Agents (ASA/MA2000)*, volume 1882 of *Lecture Notes in Computer Science*, pages 44–56, Zurich, Switzerland, September 2000. Springer-Verlag.

[78] I. Keidar and S. Rajsbaum. Open questions on consensus performance in well-behaved runs. In André Schiper, Alexander A. Shvartsman, Hakim Weatherspoon, and Ben Y. Zhao, editors, *Future Directions in Distributed Computing*, volume 2584 of *Lecture Notes in Computer Science*, pages 35–39. Springer, 2003.

[79] K. P. Kihlstrom, L. E. Moser, and P. M. Melliar-Smith. Byzantine fault detectors for solving consensus. *The Computer Journal*, 46(1), 2003.

[80] L. Lamport, R. Shostak, and M. Pease. The byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[81] J.-C. Laprie, editor. *Dependability: Basic concepts and Terminology*, volume 5 of *Dependable Computing and Fault-Tolerant Systems.* Springer-Verlag, 1992.

[82] N. Lynch. *Distributed Algorithms.* Morgan Kaufman Publishers, Inc., San Francisco, USA, 1996.

[83] P. MacKenzie, A. Oprea, and M.K. Reiter. Automatic generation of two-party computations. In *SIGSAC: 10th ACM Conference on Computer and Communications Security*. ACM SIGSAC, 2003.

[84] D. Malkhi, N. Nisan, B. Pinkas, and Y. Sella. Fairplay — A secure two-party computation system. In *Proceedings of the 13th USENIX Security Symposium*. USENIX, August 2004.

[85] D. Malkhi, F. Oprea, and L. Zhou. Ω meets paxos: Leader election and stability without eventual timely links. In *Proceedings of the 19th Symposium on Distributed Computing (DISC'05)*, volume 3724 of *LNCS*, pages 199–213, Cracow, Poland, 2005. Springer Verlag.

[86] D. Malkhi and M. Reiter. Byzantine quorum systems. *Distributed Computing*, 11(4):October, June 1998.

[87] H. Mantel. *A Uniform Framework for the Formal Specification and Verification of Information Flow Security*. PhD thesis, Universität des Saarlandes, 2003.

[88] J.-P. Martin and L. Alvisi. Fast byzantine consensus. In *Proceedings of the IEEE DSN*, pages 402–411, Jun 2005.

[89] F. Mattern. On the relativistic structure of logical time in distributed systems, March 1992.

[90] N. Mittal, F. Freiling, S. Venkatesan, and L. D. Penso. Efficient reduction for wait-free termination detection in a crash-prone distributed system. In *Proceedings of the Nineteenth International Conference on Distributed Computing*, pages 93–107. Springer-Verlag, September 2005.

[91] A. Mostéfaoui, S. Rajsbaum, M. Raynal, and M. Roy. Condition-based protocols for set agreement problems. In D. Malkhi, editor, *Distributed Computing, 16th International Conference, DISC 2002, Toulouse, France, October 28-30, 2002 Proceedings*, volume 2508 of *Lecture Notes in Computer Science*, pages 48–62. Springer, 2002.

[92] A. Mostefaoui, S. Rajsbaum, M. Raynal, and C. Travers. Irreducibility and additivity of set agreement-oriented failure detector classes. In *Proceedings of the Thirty-Seventh Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 153–162. ACM Press, July 2006.

[93] A. Mostefaoui, S. Rajsbaum, M. Raynal, and C. Travers. On the computability power and the robustness of set agreement-oriented failure detector classes. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Principles of Distributed Computing*. ACM, 2006.

[94] A. Mostéfaoui and M. Raynal. k-set agreement with limited accuracy failure detectors. In *Proceedings of the nineteenth annual ACM symposium on Principles of distributed computing*, pages 143–152. ACM Press, 2000.

[95] J.R. Munkres. *Elements Of Algebraic Topology*. Addison Wesley, Reading MA, 1984. ISBN 0-201-04586-9.

[96] G. Neiger and S. Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[97] V. P. Nelson. Fault-tolerant computing: fundamental concepts. *IEEE Computer*, 23(7):19–25, July 1990.

[98] P. G. Neumann. *Computer Related Risks*. ACM Press, 1995.

[99] H. Pagnia, H. Vogt, and F. C. Gärtner. Fair exchange. *The Computer Journal*, 46(1), 2003.

[100] P. R. Parvédy and M. Raynal. Optimal early stopping uniform consensus in synchronous systems with process omission failures. In *Proceedings of the Sixteenth Annual ACM Symposium on Parallelism in Algorithms and Architectures*, pages 302–310. ACM Press, June 2004.

[101] M. Pease, R. Shostak, and L. Lamport. Reaching agreements in the presence of faults. *Journal of the ACM (JACM)*, 27(2):228–234, April 1980.

[102] K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, March 1986.

[103] K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, 12(3):477–482, March 1986.

[104] K. J. Perry and S. Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Trans. Softw. Eng.*, 12(3):477–482, 1986.

[105] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[106] J. Rushby. Formal specification and verification of a fault-masking and transient-recovery model for digital flight-control systems. In J. Vytopil, editor, *Formal Techniques in Real-Time and Fault-Tolerant Systems 2nd International Symposium*, volume 571 of *Lecture Notes in Computer Science*, pages 237–258, Nijmegen, The Netherlands, 1992. Springer-Verlag.

[107] J. Rushby. Critical system properties: Survey and taxonomy. *Reliability Engineering and System Safety*, 43(2):189–219, 1994.

[108] M. Saks and F. Zaharoglou. Wait-free k-set agreement is impossible: The topology of public knowledge. *SIAM Journal on Computing*, 29(5):1449–1483, March 2000.

[109] N. Santoro and P. Widmayer. Time is not a healer. In *Proc. 6th Annual Symposium on Theor. Aspects of Computer Science (STACS'89)*, volume 349 of *LNCS*, pages 304–313, Paderborn, Germany, February 1989. Springer-Verlag.

[110] F. Schneider. Implementing fault-tolerant services using the state-machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

[111] P. Sousa, N. F. Neves, and P. Veríssimo. Proactive resilience through architectural hybridization. In *Proceedings of the 2006 ACM Symposium on Applied Computing*, pages 686–690. Springer-Verlag, April 2006.

[112] E.H. Spanier. *Algebraic Topology*. Springer-Verlag, New York, 1966.

[113] T. K. Srikanth and S. Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

[114] Trusted Computing Group. Trusted computing group homepage. Internet: `https://www.trustedcomputinggroup.org/`, 2003.

[115] T. Warns, F. C. Freiling, and W. Hasselbring. Consensus using structural failure models. In *Proceedings of the 25th IEEE Symposium on Reliable Distributed Systems (SRDS2006)*, pages 212–224, October 2006.

[116] D. G. Weber. Formal specification of fault-tolerance and its relation to computer security. In Sol Greenspan, editor, *Proceedings of the 5th International Workshop on Software Specification and Design*, pages 273–277, Pittsburgh, PA, May 1989. IEEE Computer Society Press.

[117] J. Yang, G. Neiger, and E. Gafni. Structured derivations of consensus algorithms for failure detectors. In *Proceedings of the Seventeenth Annual ACM Symposium on Principles of Distributed Computing*, pages 297–306. ACM Press, 1998.

[118] A. C. Yao. Protocols for secure computations. In *Proceedings of the Twenty-Third Annual Symposium on Foundations of Computer Science*, pages 160–164. Springer-Verlag, November 1982.

[119] P. Zielinski. Automatic verification and discovery of byzantine consensus protocols. In *The 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 25–28. IEEE Computer Society, June 2007.