CLASSES OF BYZANTINE FAULT-TOLERANT ALGORITHMS FOR DEPENDABLE DISTRIBUTED SYSTEMS

André Postma

ISBN 90-365-1081-3

Printed by: PrintPartners Ipskamp B.V., Enschede.

CLASSES OF BYZANTINE FAULT-TOLERANT ALGORITHMS FOR DEPENDABLE DISTRIBUTED SYSTEMS

PROEFSCHRIFT

ter verkrijging van
de graad van doctor aan de Universiteit Twente,
op gezag van de rector magnificus,
prof. dr. F.A. van Vught,
volgens besluit van het College voor Promoties
in het openbaar te verdedigen
op vrijdag 20 februari 1998 te 15.00 uur.

door

André Postma

geboren op 19 augustus 1968
te Aalst-Waalre

Dit proefschrift is goedgekeurd door de promotor:
Prof. Dr. Ir. Th. Krol

TABLE OF CONTENTS

Page

1. Introduction 1

Abstract 1

1.1. Dependable computer systems 1

1.1.1. Dependability attributes 2
1.1.2. The impairments to dependability 3
1.1.2.1. Faults, errors, and failures 3
1.1.2.2. Fault and failure classification 3
1.1.3. Dependability measures 5
1.1.4. The means of dependability 7
1.1.5. Overview 11

1.2. Fault detection techniques 11

1.2.1. Duplication 11
1.2.2. Error-detecting codes 12
1.2.3. Checksums 12
1.2.4. Self-checking and fail-safe logic 12
1.2.5. Watch-dog timers and bus timeouts 13
1.2.6. Consistency and capability checks 13
1.2.7. Processor monitoring 14
1.2.8. Program monitoring 14

1.3. Fault-tolerance techniques 14

1.3.1. Masking redundancy techniques 15
1.3.1.1. *N*-modular redundancy 15
1.3.1.2. Error-correcting codes 16
1.3.1.3. Masking logic 17
1.3.1.4. *N*-version programming 17
1.3.2. Dynamic redundancy techniques 17
1.3.2.1. Reconfigurable duplication 17
1.3.2.2. Reconfigurable *N*-modular redundancy 18
1.3.2.3. Backup sparing 18
1.3.2.4. Graceful degradation 18
1.3.2.5. Forward and backward error recovery 19

1.4. Evaluation of techniques to improve the reliability and availability of a system 20

1.5. **Overview of this thesis** 21

1.6. **References** 22

2. **A System Model** 25

Abstract *25*

2.1. **Introduction** 25

2.2. **System model** 26

2.3. **Description of the Dependable Distributed Data Storage System** 28

2.4. **Resilience to Byzantine faults** 32

2.4.1. A restricted Byzantine fault model for lock-step synchronous systems 33
2.4.2. A restricted Byzantine fault model for a distributed system in which protection against malicious intruders is required 35

2.5. **Conclusion** 37

2.6. **References** 37

3. **Authenticated Byzantine Agreement Protocols** 39

Abstract *39*

3.1. **Introduction** 40

3.1.1. Consensus protocols 41
3.1.2. Interactive consistency algorithms 43
3.1.3. Byzantine Agreement Protocols 44
3.1.4. Classification of Byzantine Agreement Protocols 47
3.1.4.1. Classification of BAPs according to the synchrony of the system 47
3.1.4.2. Classification of deterministic BAPs according to message authentication 49
3.1.5. Reduction of the communication overhead of an authenticated BAP 51
3.1.6. Guaranteeing Byzantine Agreement under less strict synchronicity assumptions 53
3.1.7. Guaranteeing interactive consistency under less strict synchronicity

assumptions 58

3.2. Authenticated Byzantine Agreement Protocols based on encoding messages into symbols of an erasure-correcting code 59

3.2.1. The Authenticated Dispersed Joined Communication algorithms 60
3.2.1.1. Definitions 61
3.2.1.2. Construction of ADJC-algorithms 64
3.2.1.3. The existence of ADJC-algorithms in the classes $A(Z_{(T,N)},K,a,N)$ 67
3.2.1.4. Behavioural properties of the ADJC-algorithms in the presence of at most $Z_{(T,N)}$ maliciously behaving processors 71
3.2.2. A class of Byzantine Agreement Protocols based on the ADJC-algorithms 74
3.2.3. The construction of authenticated Byzantine Agreement Protocols based on erasure-correcting codes 74
3.2.3.1. The Minimum Direction protocols 75
3.2.3.2. The Maximum Coding protocols 75
3.2.4. A comparison with existing BAPs 76
3.2.4.1. The criteria 76
3.2.4.2. The Lamport-protocol 76
3.2.4.3. The Dolev-protocol 76
3.2.4.4. The Minimum Direction protocol 77
3.2.4.5. The Maximum Coding protocol 78
3.2.4.6. Results 78
3.2.5. Conclusion 81

3.3. Authenticated self-synchronizing Byzantine Agreement protocols 81

3.3.1. Difference between authenticated self-synchronizing Byzantine Agreement Protocols and regular authenticated Byzantine Agreement Protocols 81
3.3.1.1. Achieving protocol synchronization between correct processors in a BAP 83
3.3.1.2. Path information 86
3.3.2. Informal description of an authenticated self-synchronizing BAP 88
3.3.2.1. Execution of an authenticated self-synchronizing BAP in a system of correctly functioning processors only 88
3.3.2.2. Execution of an authenticated self-synchronizing BAP in the presence of up to T maliciously behaving processors 90
3.3.3. The unforgeability properties 93
3.3.3.1. Valid messages 95
3.3.3.2. Generating a message 101
3.3.3.3. Relaying a message 101
3.3.3.4. Assumptions with regard to the authentication of messages 102
3.3.3.5. Satisfaction of the unforgeability properties 104
3.3.3.6. Final remarks on message authentication 109

3.3.4. Possible fault behaviour of faulty processors 110
3.3.5. Definitions with regard to the timeliness of valid messages 112
3.3.6. A description of an authenticated self-synchronizing BAP 113
3.3.7. A description of an optimized authenticated self-synchronizing BAP 116
3.3.8. Assumptions for authenticated self-synchronizing BAPs 119
3.3.9. Proofs for authenticated self-synchronizing BAPs 121
3.3.10. Proofs for optimized authenticated self-synchronizing BAPs 134
3.3.11. A comparison of authenticated self-synchronizing BAPs and optimized authenticated self-synchronizing BAPs 143
3.3.12. Conclusion 145

3.4. Authenticated self-synchronizing interactive consistency algorithms 146

3.4.1. A description of an authenticated self-synchronizing ICA 146
3.4.2. A description of an optimized authenticated self-synchronizing ICA 157
3.4.3. A comparison of authenticated self-synchronizing ICAs and optimized authenticated self-synchronizing ICAs 166

3.5. References 168

4. Distributed Cryptographic Function Application Protocols 171

Abstract 171

4.1. Introduction 172

4.1.1. Fault-tolerant data storage 172
4.1.2. Secure data storage 175
- 4.1.2.1. An introduction to cryptography 175
- 4.1.2.2. Protection of data against unauthorized reading by others 179
- 4.1.2.3. Protection of data against undetected mutilation by others 179
- 4.1.2.4. Protection of data against unauthorized reading and undetected mutilation by others 183

4.1.3. Fault-tolerant and secure data storage 184
- 4.1.3.1. Different solutions for fault-tolerant and secure data storage 184
- 4.1.3.2. Evaluation of the proposed alternatives for fault-tolerant and secure data storage 186
- 4.1.3.3. Recovery of lost or corrupted data fragments requires knowledge of the secret cryptographic function of the cryptosystem applied for signing and verification 189
- 4.1.3.4. Distributed cryptographic function application protocols 190
- 4.1.3.5. A possible implementation of the recovery service 193
- 4.1.3.6. A possible implementation of the waiting-and-synchronization process 197

4.1.4. Related work 200
4.1.4.1. Multisignature schemes 201
4.1.4.2. Secret sharing schemes 203
4.1.4.3. Threshold cryptography 205
4.1.5. Overview 207

4.2. Distributed cryptographic function application protocols based on a fault-tolerant multisignature scheme 208
4.2.1. Component cryptographic functions 209
4.2.2. Component cryptographic keys 210
4.2.3. Possible component cryptographic key distributions for DCFAPs based on a fault-tolerant multisignature scheme 211
4.2.3.1. A component cryptographic key distribution which is optimal with regard to the required number of processors 211
4.2.3.2. A component cryptographic key distribution which requires a smaller number of different component cryptographic keys and a smaller number of copies of component cryptographic keys 213
4.2.3.3. Comparison of the proposed component cryptographic key distributions 215
4.2.4. Distributed cryptographic function application 218
4.2.4.1. Assumptions 220
4.2.4.2. Description of a PCFAP based on a fault-tolerant multisignature scheme 223
4.2.4.3. Description of a DCFAP based on a fault-tolerant multisignature scheme 232
4.2.5. Efficiency considerations on PCFAPs based on a fault-tolerant multisignature scheme 237
4.2.5.1. Partial cryptographic function application protocols with completed paths of minimal length only 238
4.2.5.2. A method to reduce the number of completed paths in partial cryptographic function application protocols 240
4.2.6. Summary 243

4.3. Distributed cryptographic function application protocols based on function sharing 244

4.4. Comparison of DCFAPs based on a fault-tolerant multisignature scheme with DCFAPs based on function sharing 251

4.5. Conclusion 252

4.6. References 253

5. A Byzantine Fault-Tolerant Distributed Diagnosis Algorithm 257
Abstract *257*

5.1. Introduction 257

5.2. A distributed diagnosis algorithm with imperfect tests 258
5.2.1. Assumptions 260
5.2.2. Reduction rules 260
5.2.3. Classification of the remaining processors after having applied the reduction rules 262
5.2.4. Examples of the use of the reduction rules 263

5.3. Determining the maximum number of correct processors removed from the service by the diagnosis algorithm 264

5.4. Determining the maximum number of processors in the service that are accused of being faulty 268

5.5. Determining the minimally required number of processors in the service 269

5.6. Conclusion 269

5.7. References 269

Appendix 5.1. Proof of correctness of the reduction rules 270

Appendix 5.2. Proof of consistency of the reduction rules 274

6. Conclusion 279
Abstract *279*

6.1. Evaluation of results 279
6.1.1. Authenticated Byzantine Agreement Protocols 280
6.1.2. Distributed cryptographic function application protocols 282
6.1.3. A Byzantine fault-tolerant distributed diagnosis algorithm 283

6.2. Suggestions for further research 284

6.3. References 285

Abstract 287

Samenvatting 293

List of Publications 299

Acknowledgements 300

Biography 301

CHAPTER 1

Introduction

"Perhaps all one can really hope for, all I am entitled to, is no more than this: to write it down. To report what I know. So that it will not be possible for any man ever to say again: I knew nothing about it."

André Brink, A dry white season

Abstract

The omnipresence of computer systems in today's society creates a need for highly dependable computer systems. In this chapter, dependability and its attributes (availability, reliability, safety, confidentiality, integrity, and maintainability) are defined. Depending on the applications considered, different emphasis may be placed on the different attributes of dependability. In dependable computing, the reliability and availability of computer systems have since long played an important role. Basically, there exist four techniques to improve the reliability and availability of a computer system: fault avoidance, fault detection, masking redundancy, and dynamic redundancy. It will be argued that repairable systems should be based on masking redundancy, in case the reliability and availability improvement should be a factor 100 or more, if compared to an equivalent system (i.e., satisfying the same specifications) in which no measures are taken in order to improve the reliability or availability of the system. In this thesis, we will present several new fault-tolerant protocols that may be implemented in a distributed fault-tolerant system based on masking redundancy.

1.1. Dependable computer systems

During the past few decades, the use of computer systems has taken a high flight. The ever increasing use of computers in almost every aspect of modern life makes us more and more dependent on the correct functioning of computers.

Practical experience has shown us that failures of computer systems which perform vital functions may have serious economic consequences, or may even endanger human lives. To illustrate this, Laprie (in [Lapr94]) mentions a few recent examples of nation-wide computer-caused or computer-related failures: the 15 January 1990 telephone service breakdown in the USA, the 26-27 June 1993 credit card authorization denial in France, and the 26-27 November 1992 London Ambulance Service failure. Whereas failure of the computer system in the first two examples had primarily economic consequences, the third example shows that failures may even lead to loss of human lives. From these examples, it will be clear that in several critical applications, it is very important to minimize the probability of the occurrence of a ***system failure***.

A system failure occurs when the actual system behaviour is inconsistent with the specified system behaviour. Roughly spoken, at a system failure, the system reacts on certain input in a way, different from what the user expects. Failures are caused by faults[1] in the components or the software of the system. Faults may occur due to numerous causes, e.g., physical defects of electrical components, execution of incorrectly designed programs, or operator mistakes.

In general, avoiding a system failure is of utmost importance in either of the following cases [Aviz76]:

- the real-time delays caused by manual repair after system failure are unacceptable (e.g., in systems with hard real-time constraints, like process control applications, guidance systems, air traffic control systems, and fly-by-wire)
- it is impossible to manually repair the system (e.g., in systems that have to be unmanned, like systems used for space exploration)
- the costs of lost time and maintenance are excessively high (e.g., in banking applications, life-critical support systems, and defense systems)

Basically, it can be concluded that there exists an increasing need for so-called ***dependable computer systems***. The concept of dependability was originally introduced by Laprie in [Lapr85] in an attempt to create a consistent terminology in the field of reliable computing. ***Dependability*** is defined as that property of a computer system such that reliance can be justifiably placed on it. Clearly, this is still a rather vague definition. In general, it depends heavily on the requirements made for the application, whether or not reliance can justifiably be placed on a system.

The notion of dependability can more precisely be specified by distinguishing a number of aspects of dependability of a computer system, the relevance of which may differ for every application that is considered. In [Lapr85,Lapr92, Lapr95], Laprie has distinguished dependability attributes, impairments, measures, and means. The remainder of this section will be devoted to the aspects of dependability just mentioned.

1.1.1. Dependability attributes

In [Lapr95], ***dependability*** is defined as that property of the computer system such that reliance can justifiably be placed on the service it delivers. The ***service*** delivered by a system is its behaviour as it is perceived by its user(s). A ***user*** is another system (physical, human), which interacts with the former.

Dependability consists of the following so-called ***attributes*** (cited from [Lapr95]):

- ***availability*** (i.e., readiness for usage)
- ***reliability*** (i.e., continuity of service delivery)
- ***safety*** (i.e., non-occurrence of catastrophic consequences on the environment)
- ***confidentiality*** (i.e., non-occurrence of unauthorized disclosure of information)
- ***integrity*** (i.e., non-occurrence of improper alterations of information)
- ***maintainability*** (i.e., aptitude to undergo repairs and evolution)

Depending on the application(s) intended for the system, different emphasis may be

1. An exact definition of the terms fault, error, and failure will be given in Section 1.1.2.1.

put on the different attributes of dependability [Lapr95].

1.1.2. The impairments to dependability

Besides the attributes of dependability, Laprie also distinguishes between the so-called ***impairments*** to dependability: faults, errors, and failures. In Section 1.1.2.1., we give exact definitions of fault, error, and failure. These definitions are taken from [Lapr95]. In Section 1.1.2.2., several possible classifications of faults and failures are presented.

1.1.2.1. Faults, errors, and failures

A ***failure*** of the system occurs when the delivered service first deviates from that required by its specifications. An ***error*** is that part of the system state which is liable to lead to subsequent system failure. That means, if there is an error in the system state, then there exists a sequence of actions which can be executed by the system and which will lead to system failure, unless some corrective measures are employed [Jalo94, p.6]. The (adjudged or hypothesized) cause of an error is a ***fault***. There are numerous reasons why a fault may occur, e.g., the occurrence of a physical defect, incorrect design, operator mistakes, unstable hardware, or an unstable environment.

1.1.2.2. Fault and failure classification

Both faults and failures can be characterized according to different viewpoints. The different viewpoints, as well as the different types of faults respectively failures that can be distinguished according to any of the viewpoints, are given in Figure 1.1. The characterization of failures in Figure 1.1. is taken from [Lapr95]. The classification of faults is taken from [SiSw92, pp.22-23].

According to the viewpoint of *pattern of occurrence*, in Figure 1.1., classes of permanent, intermittent, and transient faults are distinguished. In [SiSw92, p.22], these types of faults are defined as follows. A ***permanent fault*** is a fault that is continuous and stable. In hardware, permanent faults reflect an irreversible physical change. An ***intermittent fault*** is a fault that is only occasionally present due to unstable hardware or varying hardware or software states. A ***transient fault*** is a fault resulting from temporary environmental conditions. The major difference between intermittent faults and transient faults, is that intermittent faults may be detectable and repairable by replacement or redesign, whereas transient faults are incapable of repair because the hardware is physically undamaged [SiSw92, p.22].

From the viewpoint of *origin of the faults*, physical faults and human faults are distinguished. In [SiSw92, p.22], these classes of faults are defined as follows. A ***physical fault*** is a fault which stems from physical phenomena internal to the system, such as threshold changes, shorts, opens, etc., or external changes, such as environmental, electromagnetic, vibration, etc. ***Human faults*** may be either *design faults*, which are committed during system design, modification, or establishment of operating procedures, or they may be *interaction faults*, which are violations of operating or maintenance procedures.

A different way to classify faults and failures, described in, e.g., [Cris91, BaMD93, Alst96], is to mathematically describe the ***fault behaviour*** of components, i.e., the behaviour that a component may exhibit once it is faulty. Every fault class is determined by the fault behaviour that is allowed for faults in that fault class. The advantage

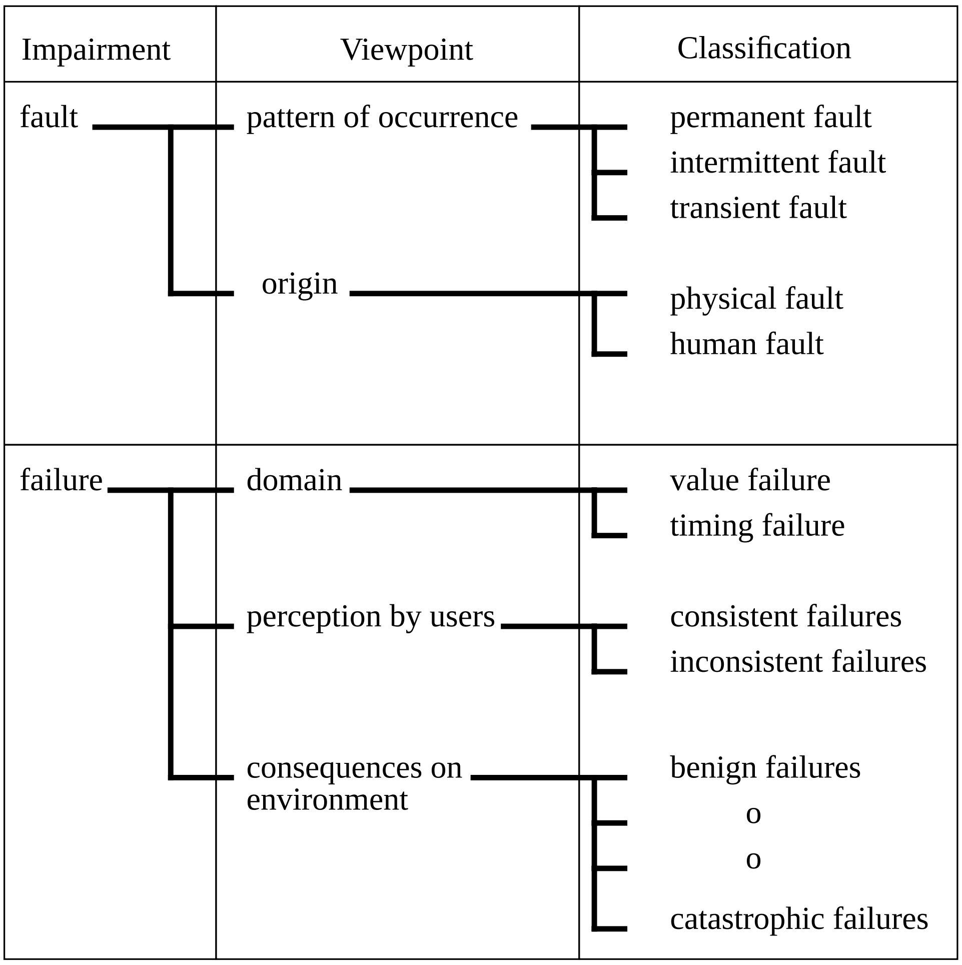


Figure 1.1. Fault and failure classification according to different view-points

of this approach is that the behaviour of the system can be mathematically described, while taking into account that several components may be faulty. Barborak et al. (in [BaMD93]) order the fault classes in a hierarchical way. The classes, from strongest to weakest, are fail-stop faults, crash faults, omission faults, timing faults, incorrect computation faults, authenticated Byzantine faults, and Byzantine faults. Each of the classes will be defined below. An interesting property is that a stronger class is a subset of a weaker class [BaMD93, p.182]. The stronger the fault model that is assumed, the easier becomes the solution that takes into account this fault behaviour [LaSP82, p.401]. The ordered fault classification depicted in Figure 1.2. is taken from [BaMD93, p.183].

The definitions of the various fault classes given below are taken from[2] [BaMD93, pp.182-183].

The class of ***fail-stop faults*** contains any fault that occurs when a processor ceases operation and alerts other processors of this fault (originally defined in [ScSc83]). The

2. Except the definition for the class of omission faults, which was taken from [Cris91].

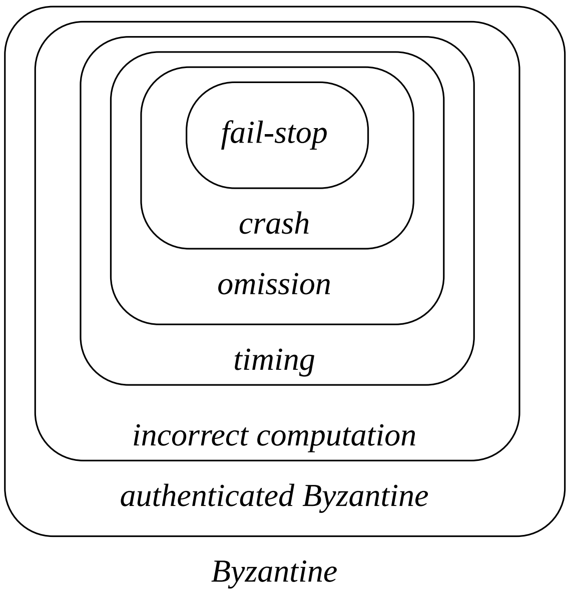


Figure 1.2. *An ordered fault classification*

class of ***crash faults*** consists of any fault that occurs when a processor loses its internal state or halts. The class of ***omission faults*** is defined as the class of faults that occur when a processor omits to respond to an input [Cris91]. The class of ***timing faults*** encompasses any fault that occurs when a processor fails to complete its task within the specified time frame, i.e. it completes its task either before or after its specified time frame or never [CASD95]. The class of ***incorrect computation faults*** contains any fault that occurs when a processor fails to produce the correct result in response to the correct inputs [LaMJ91] [3]. The class of ***authenticated Byzantine faults*** contains any arbitrary or malicious fault, with that exception that faulty processors are not capable of imperceptibly altering a message signed by a correct processor. Faulty processors may collude with other faulty processors [LaSP82]. The class of ***Byzantine faults*** contains every arbitrary or malicious fault that is possible in the system model [LaSP82]. This fault class can be considered the universal fault set [BaMD93, p.183].

1.1.3. Dependability measures

In literature, the dependability of the system can be quantified by many different functions, the so-called ***dependability measures***. A number of these dependability measures will be mentioned below. Which dependability measure is most adequate to

3. The incorrect computation fault class is a superset of the crash, omission and timing fault classes and a subset of the class of Byzantine failures. The first characteristic is true because a miscalculation may take place in time and space. Since the fault is consistent to all outside observers, though, the class of incorrect computation faults is a subset of the class of Byzantine failures [LaMJ91, BaMD93].

express the required dependability of the system, depends on the specific application that is considered.

Below, dependability measures will be presented to express the reliability and the availability of the system. Quantifications of the other dependability attributes (i.e., safety, confidentiality, integrity, and maintainability) are hardly found in literature. We will not try to quantify them here either. Whether or not these dependability attributes play an important role, is highly application-dependent.

Usually, the ***reliability*** of a system is expressed as a function $R(t)$, which is (informally) defined as the probability that the system has not suffered from a system failure until time t ($t > 0$), provided that the system initially (i.e., at time 0) works correctly [Jalo94, p.34]. The ***mean time to failure (MTTF)***, or ***expected life*** [Jalo94. p.34] of a system can be expressed in terms of $R(t)$ as follows:

$$MTTF = \int_0^\infty R(t)\,dt$$

The above dependability measures may be adequate, e.g., in ***non-repairable systems***, i.e., systems in which faulty components are not repaired, like unmanned space missions.

However, for telecommunication services, it is important that the fraction of time that the system is down is kept as small as possible, and thus, failed components should be repaired as quickly as possible. In this case, it is more adequate to express the required dependability of the system in terms of the ***availability*** of the system, i.e., the fraction of the time that the system is not out of operation [Krol91, p.17].

In [Jalo94, p.37], the ***instantaneous availability***, $A(t)$, of a component is defined as the probability that the component is functioning correctly at time t. In availability analysis, we are interested in probability at a certain instance of time. In the absence of repair or replacement, availability is simply equal to reliability. However, often, replacement is taken into account, and in this case, in availability analysis, we are interested in availability after a sufficiently long time. The ***limiting availability*** (commonly referred to as the ***availability***) of a system is the limiting value of $A(t)$ as t approaches infinity.

Krol (in [Krol91, p.17]) mentions several other dependability measures: the ***mean time between failures*** (or ***MTBF***), the ***mean time between down*** (***MTBD***), and the ***mean time to repair*** (***MTTR***). In [Krol91], these dependability measures are defined as follows.

The ***mean time between failures*** is defined as the expected length of the time interval between the moment the system is started up or a previous failure has been repaired and the moment at which a (subsequent) failure appears. The ***mean time between down*** is defined as the expected length of the time interval between the moment the system is started up (possibly after a repair) and the moment at which the system loses its functionality due to the occurrence of a fault. The ***mean time to repair*** is defined as the expected length of the time interval between system down and system up.

The ***(limiting) availability*** can be expressed as a function of *MTTF* and *MTTR* as follows [Triv82]:

$$\lim_{t \to \infty} A(t) = \frac{MTTF}{MTTF + MTTR}$$

Jalote (in [Jalo94, p.38]) notices that the above expression is not dependent on the nature of the probability distributions of lifetimes and repair times. In other words, this expression holds for probability distributions other than the exponential distribution as well.

The effect of the measures to improve the reliability of a system can be expressed in terms of a so-called ***reliability improvement factor*** [Krol91, p.18]. Let *S* be a system in which measures are taken to improve the reliability of the system, and let the resulting system be *S*'. Then, the reliability improvement factor is defined as the quotient of the *MTBD* of system *S*' and the *MTBD* of system *S*.

1.1.4. The means of dependability

This section contains a description of the means of dependability, i.e., the techniques that can be used in order to improve the dependability of a computer system, or more specifically, to improve several important attributes of the dependability of a computer system. The reliability and availability of computer systems have since long played an important role in the field of reliable computing, and many techniques are known to improve the reliability and availability of a computer system. Therefore, in this section, we will focus on techniques that can be used to improve these two attributes of dependability.

Basically, there exist four different techniques to improve the reliability and availability of a system: ***fault avoidance***, ***fault detection***, ***masking redundancy***, and ***dynamic redundancy***. Each of these techniques will be discussed below.

A direct approach to improve the reliability and availability of the system is to try to prevent faults from occurring or getting introduced in the system. This approach is called ***fault avoidance*** or ***fault prevention*** [Lapr85, Lapr92, Lapr95].

In the approach of ***fault avoidance***, a highly reliable and available system is obtained by eliminating as many faults as possible before the system is put into regular use. This is achieved by extensively testing the system software as well as the behaviour of all components needed for the system, and, while building the system, by applying only those components and software which exhibit correct behaviour during the tests. Such a system is a ***non-redundant system***, i.e., it does not contain ***redundancy*** (i.e., it does not contain components or perform checks or other computations that are superfluous as long as all components in the system function correctly). A system failure may occur as soon as a fault occurs in one or more components of the system. Although enormous improvements have been made with regard to the technical quality of computer system components, it is obvious that failures of computer systems can never be ruled out completely [Powe91, p.9]. E.g., it is known that electrical components are subject to environmental conditions (causing them to wear out) and hence, such components will sooner or later fail to satisfy their specification. Upon system failure, the system needs to be repaired. In the meantime, the system is ***down*** (i.e., not in operation). For several critical applications, like the ones described above, it is highly important that the probability that a system failure occurs is being made as small as possible.

By means of applying fault prevention techniques, the reliability of the system can be improved by a factor 10 without exceptionally high costs [Krol91, p.18].

In order to improve the reliability and availability of the system beyond that what can be reached by applying fault prevention techniques, one should not only try to prevent faults from occurring or being introduced in the system, but, in addition, the system should be designed in such a way that the occurrence of faults during operation is taken into account. Thus, besides fault prevention techniques, techniques of ***fault detection***, ***masking redundancy*** (also called ***static redundancy***), and/or ***dynamic redundancy*** should be applied in the design of a dependable system. Application of either of these techniques requires the system to be a so-called ***redundant system***, i.e., a system in which redundancy is applied. Whereas systems based merely on fault detection and/or fault avoidance can only guarantee that the specified service is provided until a fault occurs, systems based on masking redundancy and/or dynamic redundancy may even continue to provide the correct service despite the occurrence of one or more faults in the system. The latter systems are therefore often called ***fault-tolerant systems***[4]. A schematic overview of dependable systems (analogous to [SiSw92, p.83]) is given in Figure 1.3.

In ***systems based on fault detection***, redundancy is applied in order to detect faults. Usually, upon detection of faults, the system reports occurrence of a fault, after which the system is brought down, diagnosed, and manually reconfigured to allow a restart [SiSw92, p.82]. Some hardware techniques used for fault detection are duplication, error-detecting codes, checksums, self-checking and fail-safe logic, watch-dog timers and bus timeouts, consistency and capability checks, and processor monitoring. A software technique used for fault detection is program monitoring. Each of these techniques will briefly be discussed in Section 1.2. For more details on these techniques, see, e.g. [SiSw92, pp. 96-138].

In ***fault-tolerant systems***, redundancy is applied in order to tolerate faults. I.e., the system contains components (***hardware redundancy***), or performs checks or other computations (***software redundancy***) which are not required (i.e., redundant) as long as all components of the system function correctly. The redundancy aims to enable the system to continue to deliver the specified system service despite the presence of faulty components in the system. This is in contrast with systems based on fault avoidance or fault detection techniques, in which a failure of a system component may lead to a system failure.

In most fault-tolerant systems, redundancy is applied in the form of both software

4. Systems that are based on *fault detection* techniques are usually **not** regarded as *fault-tolerant* systems [SiSw92, p.83], since these systems are not capable of tolerating any type of fault. In particular, a system based on fault detection techniques cannot tolerate permanent faults, although permanent faults may be detected by such a system. Transient and intermittent faults (see Section 1.1.2.2.) may only be tolerated by using retry techniques.

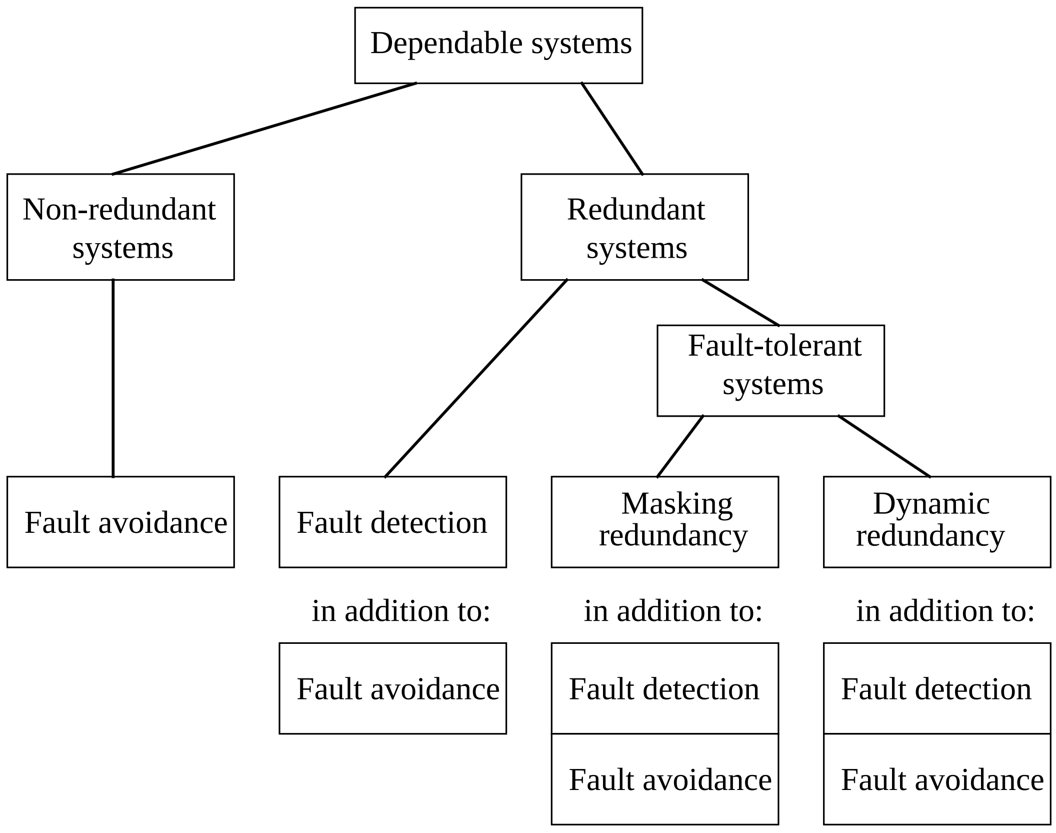


Figure 1.3. Schematic overview of dependable systems

redundancy (or time redundancy) and hardware redundancy (or component redundancy). Systems not containing hardware redundancy can *not* be made resilient to ***permanent component failures***[5]. By applying hardware redundancy, a system can be made resilient to permanent component failures.

In ***systems based on masking redundancy***, redundancy is applied in order to mask failures of components of the system. The system may continue to provide the specified service as long as the number of component failures stays below a predefined maximum. Several hardware techniques used in systems based on masking redundancy are, e.g., *N*-modular redundancy, error-correcting codes, and masking logic. A software technique is *N*-version programming. A short investigation of each of these techniques will follow in Section 1.3.1. For more details, see, e.g., [SiSw92, pp. 138-169, and pp. 206-213].

In ***systems based on dynamic redundancy***, upon component failures, the system is dynamically reconfigured so as to on-line replace the failed components by correctly functioning spare components. Several hardware techniques used in systems based on dynamic redundancy are, e.g., reconfigurable duplication, reconfigurable *N*-modular redundancy, backup sparing, and graceful degradation. Software techniques are, e.g.,

5. See Section 1.2.2.

backward and forward error recovery. Section 1.3.2. provides a short introduction to each of these techniques. For more details, see, e.g., [SiSw92, pp. 169-201, and pp. 213-219].

In order to improve the reliability and availability of a system, often, a combination of techniques is used, i.e., the design of a dependable system involves application of techniques of fault-avoidance, fault-detection and fault-tolerance.

In a redundant system, treatment of faults that occur in the system may consist of a number of steps, discussed below. In these steps, the above-mentioned techniques of fault-detection and fault-tolerance may be used as indicated below.

According to [SiSw92, pp.80-82], ***fault treatment*** consists of the following steps:

1. ***Fault detection***
   In this initial step, it is detected that something unexpected has happened in the system. Many techniques to detect faults exist (see Section 1.2.). In general, an arbitrary period of time, called the ***fault latency***, has passed between the occurrence and the detection of a fault. It is even possible that some faults (e.g., transient faults, see Section 1.1.2.2.) are never detected.

2. ***Fault diagnosis***
   In this step, diagnosis is performed in order to obtain information about the location and/or the nature of the detected fault. This step is necessary if the fault detection technique does not provide this information.

3. ***Reconfiguration***
   This step occurs in case the detected fault has led to a permanent component failure. In a system based on dynamic redundancy, reconfiguration may be done on-line by reconfiguring the system such as to replace the failed component by a backup spare. Alternatively, the failed component may be switched off and the system capability degraded in a process called ***graceful degradation*** ([SiSw92, p.81]). In a system which is not based on dynamic redundancy, reconfiguration has to be done off-line. In a system based on masking redundancy, the system may be able to continue its specified service despite the presence of faults. If a critical task is performed, reconfiguration may be postponed until the critical task has finished.

4. ***Recovery***
   In this step, the effects of faults are eliminated, either by means of techniques of fault masking or retry. Fault masking techniques hide the effects of component failures by having redundancy provide for sufficient correct information. In retry techniques, in case the occurrence of a fault inhibits execution of a certain operation, it is tried multiple times to execute this operation. For transient or intermittent faults (see Section 1.1.2.2.), this approach may be successful.

5. ***Restart***
   Once recovery has been finished, the system can be restarted. A ***hot restart*** (i.e., a resumption of all operations from the point of fault detection ([SiSw92, p.81]) is

only possible if the fault did not cause permanent damage. Otherwise, a ***warm restart*** (i.e., some of the processes can be resumed without loss) or a ***cold restart*** (i.e., a re-initialization of the system) is required.

6. ***Repair***
   In this step, a failed component is replaced. In systems based on masking redundancy, the component can be replaced on-line (i.e., without interrupting system operation) if for a sufficiently long period of time, the component is not necessary for operation, e.g., if the component is temporarily idle. Otherwise, the component has to be replaced off-line. In systems based on dynamic redundancy, the component may be replaced on-line by a backup spare. In other systems, repair has to be done off-line.

7. ***Reintegration***
   In this step, the repaired component must be reintegrated into the system. For on-line repair, reintegration must take place without interrupting system operation.

In *non-redundant systems*, only fault avoidance techniques can be applied, and human intervention is required in all steps of fault treatment given above. In *redundant systems*, it is also possible to apply techniques of fault detection, masking redundancy and / or dynamic redundancy. In such a system, one or several of the fault treatment steps can be automated.

Since redundancy is applied in fault-tolerant systems, these systems are usually more expensive (in terms of hardware and/ or computation time) than systems which provide the same functionality but which are not fault-tolerant. The more reliable and available we want to make the system, the more redundancy is required, and, inherently, the more expensive the system will be. In practice, the design of a fault-tolerant system is mainly determined by the allowed system costs and the performance, reliability and availability that is required.

In this thesis, our focus will be on the design of algorithms that are used in fault-tolerant systems, i.e., systems in which ultra-high dependability is required.

1.1.5. Overview

The rest of this chapter is structured as follows. Section 1.2. gives a more detailed discussion of the fault avoidance techniques mentioned in Section 1.1.3. In Section 1.3., the previously mentioned fault-tolerance techniques are investigated. Section 1.4. evaluates the techniques of Section 1.2. and 1.3. Finally, Section 1.5. provides an overview of the rest of this thesis.

1.2. Fault detection techniques

In this section, the fault detection techniques, as mentioned in Section 1.1.3., will subsequently be discussed.

1.2.1. Duplication

The hardware technique of duplication can be applied at every level of design (i.e., at component level or system-level). Duplication of a component (respectively, a system) consists of simply making a copy of the component (respectively, the system). Both copies have the same functionality. When a failure occurs in one of the copies, the two copies will no longer be the same, and a simple comparison will detect the fault. An advantage of using duplication is, that duplication successfully detects all single faults except that of the comparison element [SiSw92, p.97]. It is even possible to detect double faults, provided that the output of both copies is different. A disadvantage is that the method requires much redundancy. The cost of duplication is twice that of an equivalent simplex system, plus the cost of the comparison element [SiSw92, p.101]. The technique of duplication is applied in the commercially available systems from Tandem and Stratus.

1.2.2. Error-detecting codes

Error-detecting codes consist of systematically adding redundancy to information. Information is encoded into so-called ***code words***, which are simply fixed-length sequences of bits. The code words contain redundant bits, which enables the set of code words of a certain length l (for some $l > 1$) to be only a subset of the set of all possible words of length l, i.e., all possible sequences of bits of length l. Mutilations of code words can be detected, provided that the resulting word is not a code word. Many different coding techniques and error-detecting codes exist. An introduction to this subject can be found in [MaSl78, VaOo89].

1.2.3. Checksums

A cheap method of fault detection is checksumming. In fact, checksums are a form of error-detecting codes. The following description of a checksum is taken from [SiSw92, p.115]. The checksum for a block of s words is formed by adding together all of the words in the block modulo-n, where n is arbitrary. The block of s words and its checksum together constitute a code word in a so-called ***linear separable code***, i.e., a code in which the code words are formed by a concatenation of data bits and redundant bits, such that the two sequences of bits can be easily distinguished. The number of bits in the sum is usually limited. This quantity is compared with the checksum which was formed and stored at the moment the block was transmitted for the last time. Checksumming is cheap in terms of required redundancy. However, it has a number of disadvantages. Main disadvantages are, that it takes a long time to detect faults (s additions and a comparison), the checksum must be updated on each write operation on the block, and the diagnostic resolution is low. Viz., in memories, the detected fault could be in the block of s words, the stored checksum, or the checking circuitry, whereas, in data transmission, the fault could be in the data source, the transmission medium, or the checking circuitry [SiSw92, p.115].

1.2.4. Self-checking and fail-safe logic

A disadvantage of the previously presented methods of duplication, error-detecting codes and checksums is that they all contain a ***single point of failure***, i.e., a specific component in the system is not allowed to fail. The techniques of duplication and checksumming are vulnerable to a failure in the comparison element, whereas the technique of error-detecting codes is vulnerable to a failure of the decoder. In order to avoid such single point of failures, self-checking logic has been introduced. See [SiSw92, p.124-130] for an overview.

In [AnMe73, SiSw92], ***totally self-checking logic*** is defined as logic which is both self-testing and fault-secure. A ***self-testing*** circuit is defined as a circuit in which, for every fault from a prescribed set, the circuit produces a noncode output for at least one code input [Ande71, SiSw92]. A circuit is called ***fault-secure*** if, for every fault from a prescribed set, the circuit never produces an incorrect output for code inputs. A disadvantage of application of totally self-checking logic is that much redundancy is required.

Alternatives in which fewer redundant elements are required, are partially self-checking logic, and fail-safe logic.

A circuit is ***partially self-checking*** if it is self-testing for a set N of normal inputs and a set F_t of faults, and it is fault-secure for a set I (a nonempty subset of N) and a set F_s of faults [SiSw91, p.126]. In partially self-checking logic, the required component redundancy is lower than that of duplication, be it, however, at the cost of an increased fault latency (i.e., undetected faults may exist for a while prior to detection).

A circuit is ***fail-safe***, if, for every fault from a prescribed set, any input produces a ‘safe’ output, i.e., one of a preferred set of erroneous outputs [SiSw92, p.130]. Fail-safe techniques are thus not concerned with the detection of faults per se, and they can result in lower redundancy costs than self-checking techniques.

A disadvantage of applying the technique of self-checking or fail-safe logic is that it relies on only one of the gates in the circuit being defective. If all gates are integrated in an IC, this assumption applies to only a small percentage of possible faults [Krol91, p.23].

1.2.5. Watch-dog timers and bus timeouts

Watch-dog timers are used to keep track of the execution time needed by the system to perform a certain task. The timer is set before execution of a certain task by the system, indicating the maximum time that is allowed to perform this task. After having executed this task, the timer is reset. If the system works properly, it is always possible to timely reset the timer. However, if the system fails in some way, it may also fail to reset the timer, which may be detected as soon as the timer expires. Of course, the system may only partially fail, produce erroneous output and still be able to correctly reset its watch-dog timer. Since output data is not checked, not every failure is detected. However, the method is very effective, since many tasks fail in an infinite loop [Krol91, p.24], hence they fail to reset the watch-dog timer, and the failure can be detected.

In the method of ***bus timeouts***, time limits are set for certain responses required by the bus protocol. E.g., when one device requires a response from another device, a failure to respond in time (which may indicate a possible failure) will be detected.

1.2.6. Consistency and capability checks

Consistency checks verify if inputs or computed results are reasonable. A simple example is a ***range check***, i.e., a check whether an input or a computed result is within a valid range. Most computers use some form of consistency checking.

In ***capability checking***, access to objects is limited to users with the proper authorization. ***Objects*** include memory segments and I/O devices; ***users*** might be processes or independent physical processors in a system [SiSw92, p.133]. One can think of read/ write protection of files as it is done in the UNIX operating system. Another method of capability checking is the use of passwords.

1.2.7. Processor monitoring

Besides duplication, ***processor monitoring*** may be an efficient way to detect logic control failures and check standard microprocessors [SiSw92, p.134]. Processor monitoring techniques are classified according to the information monitored: control-flow checking techniques and assertion checking techniques.

Control-flow monitoring techniques detect sequence errors, i.e., errors which cause a processor to jump to an incorrect next instruction. ***Assertion checking techniques*** make use of properties of program data by periodically checking for program data. Assertion checking requires the user to identify invariant properties of program data and devise code that will check for these properties [SiSw92, p.136]. Examples of invariant properties include cases in which a value of a variable should always be within a particular range, the output value values of a function are related to the input values by the inverse of that function, and variable values in a set increase or decrease monotonically. Success of this method depends heavily upon the existence of invariants in an application.

1.2.8. Program monitoring

This software fault-detection technique consists of monitoring the execution of a piece of software step-by-step. For this purpose, so-called ***debuggers*** can be applied.

1.3. Fault-tolerance techniques

Two different fault-tolerance techniques can be distinguished: masking redundancy techniques and dynamic redundancy techniques. These techniques will be the subject of Section 1.3.1. and 1.3.2., respectively.

In systems based on *masking redundancy*, fault detection, fault localization and recovery are automatically performed by the system, and form a whole. In systems based on masking redundancy, failures are tolerated and do not cause system operation to be interrupted. Masking redundancy is used, e.g., in computers with error-correcting code memories or with majority-voted redundancy in a fixed configuration (i.e., the logical connections between circuit elements remain constant) [SiSw92, p.83].

In systems based on *dynamic redundancy*, it is possible to automatically perform fault detection, fault localization, reconfiguration, recovery and restarting. Dynamic redundancy covers those systems whose configuration can be dynamically changed in response to a fault, or in which masking redundancy, supplemented by on-line fault detection, allows on-line repair. Examples include multiprocessor systems that can degrade gracefully in response to processing element failures and triplicated systems that are designed for on-line repair [SiSw92, pp.83-84].

1.3.1. Masking redundancy techniques

In this section, the masking redundancy techniques mentioned in Section 1.1.3. are subsequently discussed.

1.3.1.1. *N*-modular redundancy

The concept of *N*-modular redundancy can be applied at all levels of system design. In *N*-modular redundancy, N ($N > 2$) copies of the original system (respectively circuit) are placed in parallel, and a majority vote is taken over the outputs of the copies. In this way, faulty behaviour of up to $\lfloor N / 2 \rfloor$ copies can be masked. Here, for any nonnegative integer value x, $\lfloor x \rfloor$ is defined as the largest integer less than or equal to x. Usually, N is chosen to be odd in order to avoid the uncertain state in which the output is a tie.

A well-known example of *N*-modular redundancy applied at system level is the so-called TMR-system (Triple Modular Redundant-system). In a TMR-system, the original system is triplicated (i.e., $N = 3$), and a majority vote is taken over the outputs. The majority vote is taken by a voter, which must also be triplicated in order to avoid a single point of failure. In Figure 1.4., a schematic representation of a TMR-system is given. It is easy to see that a failure in one of the three copies can be masked. The

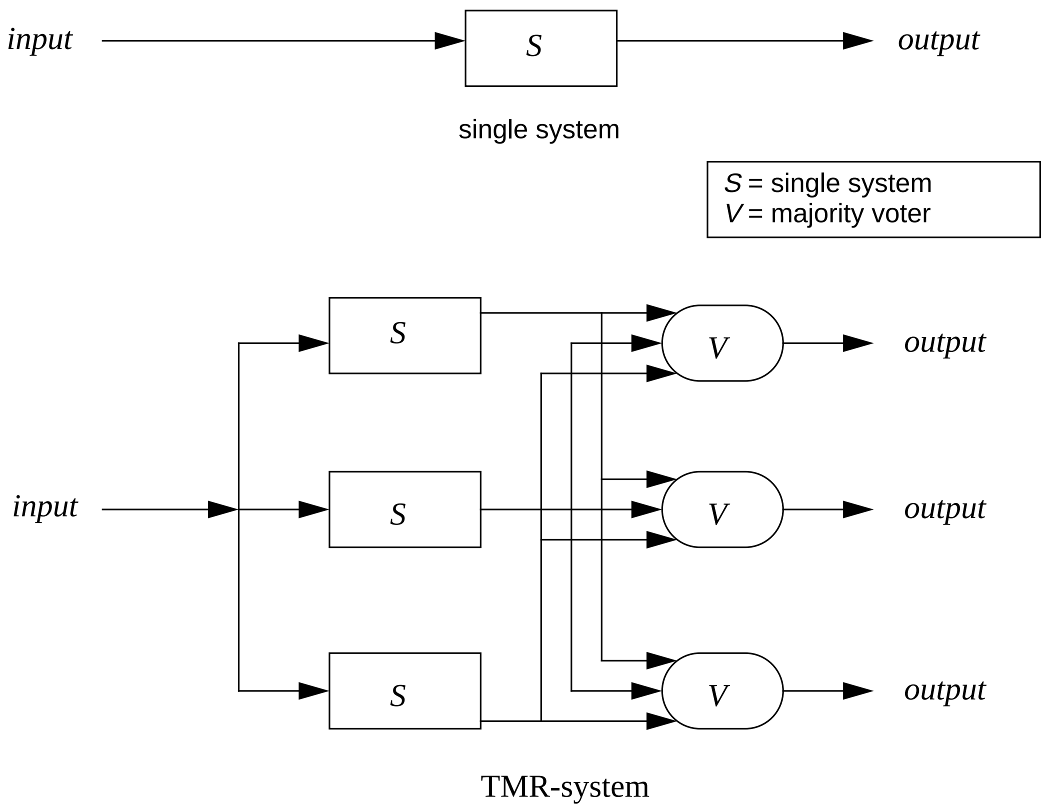


Figure 1.4. Schematic representation of a TMR-system

masking is accomplished by means of a majority (two-out-of-three) vote on the system outputs.

Krol (in [Krol91, p.25]) mentions that, in general, the voters will be provided with additional hardware which records the fact that a fault has been masked and in which module it has occurred. In this way, fault localization can be automatically performed by the system.

Krol (in [Krol91, p.25-26]) also notices that the addition of redundancy must be done very carefully. Extra hardware increases the probability of defects, as a result of which the reliability may decrease instead of increase. To see this, consider the TMR-system sketched above. If a fault causes a failure of more than one of the three copies, the TMR-system will break down. It is therefore important to minimize the probability that such ***dependent faults*** occur. An approach to solve this problem is to subdivide the system into so-called ***fault isolation areas*** or ***fault containment units***. Faults within a fault isolation area may be dependent, but faults in different fault isolation areas are independent.

However, for a TMR-system, it is very difficult to create a fault isolation area for each of the three subsystems, since, in practice, clock generators, supply voltages, and electromagnetic pulses may be the cause of dependent faults, causing more than one subsystem to fail. In [Krol91, p.27], Krol shows that dependent faults may drastically decrease the reliability improvement factor.

1.3.1.2. Error-correcting codes

Error-correcting codes are the most commonly used means of masking redundancy [SiSw92, p.146]. Just like error-detecting codes, error-correcting codes are a means of systematically adding redundancy to information. Information is encoded into code words. The code words contain redundant bits in such a way that a limited number of faulty bits can always be corrected. How many faulty bits in a code word can be corrected depends on the so-called ***Hamming distance*** of the applied error-correcting code.

The Hamming distance of a code is the minimum number of bit positions in which any two words in the code differ [Jalo94, p.23]. Let d be the Hamming distance, D be the number of bit errors that can be detected, and C be the number of bit errors it can correct, then the following relation is always true [Jalo94, p.23]:

$$d = C + D + 1, \text{ with } D \geq C$$

This means that all codes with Hamming distance $d > 2$ can be applied for both error-correction and error-detection. Notice that, in the above relation, correction of bit errors implies that these errors are detected.

Let k be the number of symbols in a data word, and n be the number of symbols in a code word. From [MaSl78], we know that an error-correcting code capable of correcting up to T errors can be constructed if and only if $n \geq k + 2T$. From [Krol91], we know that such a code always exists if the symbol size b satisfies $b \geq {}^{2}\log(n\text{-}1)$.

In regular binary codes like ASCII, the Hamming distance is 1 (i.e., the code contains code words that differ in only one bit position), hence, $D = 0$ and $C = 0$ (i.e., no error-detection and no error-correction is possible). Addition of a parity bit to each code word increases the Hamming distance to 2 (viz., if two original code words differ in one position, then the parity bits of these code words will also be different, creating a Hamming distance of 2). Codes with Hamming distance 2 can only detect single bit errors (i.e., $D = 1$).

A special kind of error-correcting codes are the so-called ***erasure-correcting codes***. These codes are able to correct a number of ***erasures***, i.e., erroneous bits of which the position in the code word is known. Let k be the number of symbols in a data word, and n be the number of symbols in a code word. From [VaOo89], we know that an erasure-correcting code capable of correcting up to T erasures can be constructed if and only if $n \geq k + T$. From [Krol91], we know that such a code always exists if the symbol size b satisfies $b \geq {}^{2}\log(n\text{-}1)$. An example in which an erasure-correcting code can be used is a bus with a single parity bit in which a particular bit line is known to be failed.

Many different error-correcting codes have appeared in literature. For an overview, see, e.g., [MaSl78, VaOo89].

1.3.1.3. Masking logic

In contrast with the previous two techniques, ***masking logic*** includes fault masking at gate level. Masking logic usually requires massive use of redundant gates, hence, only few of the techniques are actually used [SiSw92, p.162]. All techniques employ redundant inputs to each gate. In this section, we will not further investigate masking logic, because of its sporadical use. For details about masking logic, see, e.g., [SiSw92, pp.161-169].

1.3.1.4. *N*-version programming

The concept of ***N-version programming*** (also called ***diverse programming*** or ***design diversity***) consists of independently developing N versions of a program. The software versions are developed by independent design teams utilizing different design methodologies, algorithms, compilers, run-time systems, and hardware components. An advantage is that the entire system may be more reliable than any single independently developed copy, although it is in general impossible to quantify the reliability improvement. The disadvantages are additional design costs, costs of concurrent execution, and potential sources of dependent faults. For more details, see, e.g., [SiSw92, pp.207-213].

1.3.2. Dynamic redundancy techniques

Dynamic redundancy techniques involve the reconfiguration of system components in response to failures [SiSw92, p.169]. The reconfiguration prevents failures from influencing system operation. In all dynamic redundancy techniques, reconfiguration and recovery are automatically performed by the system.

1.3.2.1. Reconfigurable duplication

The technique of duplication discussed in Section 1.2.1. can be used for fault detection. However, a duplicated system is not fault tolerant, unless some adaptations are made to the system. The resulting system is called a ***reconfigurable duplicated system***. In such a system, additional hardware is needed in order to detect which module is faulty in case the outputs of both systems are different. Upon localization of the faulty module, the system should be reconfigured, i.e., the faulty module is disconnected from the system, and the comparison element is disabled. The resulting simplex system still satisfies the system specification. For more details, see [SiSw92, p.171-174].

1.3.2.2. Reconfigurable *N*-modular redundancy

The technique of *N*-modular redundancy discussed in Section 1.3.1.1. can be applied to tolerate a number of faulty modules. However, as discussed in Section 1.3.1.1., an *N*-modular redundant system is not capable of tolerating more than $\lfloor N / 2 \rfloor$ faulty modules. By making some adaptations to the NMR-system, it is possible to have the system tolerate more than $\lfloor N / 2 \rfloor$ faulty modules. The resulting system is called a ***reconfigurable N-modular redundant system***. Basically, two different types of reconfigurable NMR-systems exist: systems based on ***hybrid redundancy***, and systems based on ***adaptive voting***. Both types of systems will be briefly discussed below.

In systems based on *hybrid redundancy*, the technique of *N*-modular redundancy is combined with the technique of backup sparing (which will be discussed in the next section). The system is assumed to consist of (N+S) modules, *N* of which are active and *S* of which are passive (spare) modules. The system output is equal to the majority vote taken over the outputs of the *N* modules. If the outputs of the *N* modules are not all equal, but the output values of a majority of modules are still equal, then these modules are assumed to be correct, and the other modules in the system are replaced by spare modules. As long as the number of faulty modules in the set of *N* active modules does not exceed $T = \lfloor N / 2 \rfloor$ before reconfiguration can take place, the system can tolerate the failure of up to $P = (T + S)$ of its modules [SiSw92, p.175]. More details can be found in [SiSw92, pp.174-178].

Systems based on *adaptive voting* are *N*-modular redundant systems, in which the system output is determined by the weighted value of each of the output values of the modules. The weight factors are usually zero or one. For more details, see, e.g., [SiSw92, pp.178-182].

1.3.2.3. Backup sparing

A system based on backup sparing consists of (N+S) modules, *N* of which are active and *S* of which are passive modules. The system output is determined by taking a majority vote over the outputs of the *N* active modules. Upon failure of an active module, it is replaced by a spare module. The concept of backup sparing can also be combined with techniques other than *N*-modular redundancy. More details can be found in [SiSw92, pp.187-190].

1.3.2.4. Graceful degradation

The technique of ***graceful degradation*** differs from the previously discussed dynamic redundancy techniques in that in a gracefully degradable system, the redundant (spare) modules are also used for normal system operation, while in reconfigurable duplication, reconfigurable *N*-modular redundancy, and backup sparing, the spare modules only perform useful work, if they have replaced a failed active module. As a consequence, upon failure of one or more modules of a gracefully degradable system, the performance and / or the functionality of the system will degrade, since the system function must in this case be carried out by fewer modules.

1.3.2.5. Forward and backward error recovery

Forward and backward error recovery are the two major *software* dynamic redundancy techniques.

Forward error recovery (or ***roll-forward***) attempts to continue operation with the current system state [SiSw92, p.213]. Forward error recovery is highly application-dependent and can only occasionally be applied, e.g., in a real-time control system in which a missed response to a sensor input can be tolerated.

Backward error recovery (or ***roll-back***) uses previously saved correct state information as the starting point after a failure [SiSw92, p.213]. Basically, there exist four forms of backward error recovery: retry, checkpointing, journaling, and recovery blocks, each of which will be briefly discussed below.

Upon detection of an error, ***retry techniques*** aim at repairing the system, after which the operation affected by the error is retried [SiSw92, p.200]. If the error is *transient*, repair consists in pausing long enough for the transient error to disappear. If the error is *permanent*, then the system should be reconfigured. Reconfiguration can only remove the error, if it is possible to recover the system state before occurrence of the error.

Checkpointing techniques consist of saving some subset of the system state at specific points (so-called checkpoints) during process execution. The information that must be stored is the subset of the system state (i.e., data, programs, machine state) that is necessary to the continued successful execution and completion of the process past the checkpoint [SiSw92, p.214]. Upon detection of an error, the system is repaired and the system and the process state is reset to the state stored in the latest checkpoint.

The ***journaling technique*** consists of making a copy of the initial state of the system, and storing all transactions that change the state of the system [SiSw92, p.216]. Upon detection of an error, the system can recover from this error by again starting from the initial state, and running through the transactions a second time. Journaling is considered to be a simple but very inefficient backward error recovery technique.

Recovery blocks combine elements of checkpointing and backup alternatives, as such providing resilience to both software design faults and permanent and transient hardware faults [SiSw92, p.216]. The method minimizes the amount of state information backed up and releases the programmer from determining which variables should be checkpointed, and when. Each recovery block contains the variables global to that block that will be automatically checkpointed to a recovery cache if they are altered within the block. It is assumed that within a recovery block it is possible to choose among a set of different design alternatives (i.e., different versions of a program, obtained by applying the technique of *N*-version programming, described in Section 1.3.1.4.). Upon entry of a recovery block, the primary alternative is executed and, after that, subjected to an acceptance test in order to detect any errors in the result. If the test is passed, then the block is exited. If the primary alternative fails to pass the test, then the contents of the recovery cache pertaining to this recovery block are reinstated, and the next alternative is invoked, etc., until the test is passed or no alternatives are left, in which case an error is reported.

1.4. Evaluation of techniques to improve the reliability and availability of a system

In the previous sections, a number of techniques have been presented, that can be applied in order to improve the reliability and availability of a system. In this section, we will evaluate the applicability of the proposed techniques, in the case that ultra-high reliability and availability is required. Analogously to [Krol91, p.29-31], it will be argued that any repairable system in which a reliability improvement of at least a factor 100 is required, should be based on masking redundancy.

Recall from Section 1.1.3. that four basic techniques to improve the reliability and availability of a system can be distinguished: fault avoidance, fault detection, masking redundancy, and dynamic redundancy.

As already stated in Section 1.1.3., by applying *fault avoidance techniques*, it is possible to improve the reliability and availability of a system by a factor 10. For systems in which such a reliability and availability improvement is sufficient, it is not very cost effective to apply redundancy in the system [Krol91, p.29].

Notice that mere application of *fault detection techniques* does not improve the reliability or availability of a system. Hence, fault detection techniques should always be used **in combination with** other techniques in order to improve the reliability and availability of a system.

If the required reliability and availability improvement lies beyond that what can be reached by applying fault avoidance techniques, it is necessary to use *techniques of masking and / or dynamic redundancy.*

Although systems based on techniques of dynamic redundancy usually require less redundancy than equivalent systems based on masking redundancy, dynamically redundant systems have a number of disadvantages (mentioned below), as a result of which systems based on masking redundancy are to be preferred to dynamically redundant systems.

As stated in Section 1.3., in a dynamically redundant system, usually, the steps fault detection, fault localization, reconfiguration, recovery and restarting are automatically performed. This poses the following requirements on the system:

- ❏ Fault detection techniques should be applied in order to be able to detect faults.
- ❏ Upon fault detection, the application program is interrupted immediately, and a diagnosis algorithm is run in order to locate the system component(s) liable for the fault. After that, reconfiguration of the system is done automatically and on-line.
- ❏ Finally, recovery software is required in order to perform the recovery process, after which a restart can take place.

A disadvantage of such a system is that it is not always desirable or even allowed to immediately interrupt the application program executed by the system upon detection of a fault. Furthermore, the recovery software will usually interfere with the application software, as such increasing the complexity of the application software. Finally, the reliability and availability improvement depends highly on the so-called ***coverage*** (originally introduced in [BoCS69]), i.e., the ability to recover from a fault [Krol91, p.30]. The coverage is determined by the effectiveness of the diagnosis algorithm and the recovery software. Diagnosis algorithms often fail to find transient and intermittent faults.

All the above disadvantages can be circumvented by applying masking redundancy techniques instead of dynamic redundancy techniques in order to improve the reliability and availability of a system.

A main disadvantage of applying dynamic redundancy techniques is that upon detection of a fault, the application program is immediately interrupted. However, in order to limit the effects of a fault, it is advantageous to replace failed processors as soon as possible. So, it would be desirable to perform the steps fault localization, reconfiguration, recovery and restarting as soon as the application program has finished or may be interrupted. In this way, the recovery software need not interfere with the application software, or cause an immediate interrupt of the application program upon detection of a fault.

Summarizing, it may be concluded that any repairable system in which a reliability and availability improvement of at least a factor 100 is required, should be based on masking redundancy. In such a system, there exists a need for diagnosis and recovery software, in order to be able to timely recover from faults. Execution of this recovery software should, however, not interfere with application programs executed by the system.

1.5. Overview of this thesis

In this thesis, a number of new fault-tolerant algorithms will be introduced, which are applicable in a distributed computer system based on masking redundancy. The applied system model will be the subject of Chapter 2. In that chapter, we will also present a vehicle, called the Dependable Distributed Data Storage System, in which all new algorithms presented in Chapters 3, 4, and 5, can conveniently be applied.

Chapter 3 presents several new solutions to the well-known Byzantine Agreement Problem. This problem is of key importance in the field of reliable computing, and has originally been formulated by Lamport et al. in [LaSP82]. A solution to this problem aims to guarantee interactive consistency, informally, agreement on input values within the correct processors in the system. In contrast to existing solutions to the problem, the solutions in this thesis are applicable in a distributed system, with uncertain message delivery times and arbitrary clock skew between the clocks of correct processors in the system.

Chapter 4 introduces the so-called distributed cryptographic function application protocols. These protocols can be conveniently applied in a recovery process, in order to recreate lost or corrupted data in a data storage system.

In Chapter 5, a new distributed diagnosis algorithm is introduced, by means of which it is possible to remove all detectably faulty processors from a group of processors, on basis of test results that the processors have obtained from each other.

Finally, Chapter 6 contains conclusions as well as some suggestions for future research.

1.6. References

[Alst96] Alstein, D., **Distributed Algorithms for hard real-time systems**, Ph.D. thesis, Eindhoven University of Technology, 1996.

[Ande71] Anderson, D.A., **Design of Self-Checking Digital Networks Using Code Techniques**, Ph.D. thesis, University of Illinois at Urbana-Champaign, 1971.

[AnMe73] Anderson, D.A., and Metze, G., Design of Totally Self-Checking Check Circuits for m-out-of-n Codes, in: **IEEE Transactions on Computers**, Vol. C-22, No. 3, March 1973, pp.263-269.

[Aviz76] Avizienis, A., Fault-Tolerant Systems, in: **IEEE Transactions on Computers**, Vol. C-25, No.12, December 1976, pp. 1304-1312.

[BaMD93] Barborak, M., Malek, M., and Dahbura, A., The Consensus Problem in Fault-Tolerant Computing, in: **ACM Computing Surveys**, Vol.25, No.2, June 1993, pp.171-220.

[BoCS69] Bouricius, W.G., Carter, W.C., and Schneider, P.R., Reliability modeling techniques of self-repairing computer systems, in: **Proceedings of the 24th National Conference of the ACM**, 1969, pp.295-333.

[CASD95] Cristian, F., Aghili, H., Strong, R., and Dolev, D., Atomic broadcast: From simple message diffusion to Byzantine Agreement, in: **Information and Computation**, Vol. 118, 1995, pp. 158-179. (An earlier version of this paper appeared in: **Proceedings of the 15th International Symposium on Fault-Tolerant Computing**, Ann Arbor, June 1985, pp. 200-206).

[Cris91] Cristian, F., Understanding Fault-Tolerant Distributed Systems, in: **Communications of the ACM**, Vol.34, No.2, February 1991, pp. 56-78.

[Jalo94] Jalote, P., **Fault Tolerance in Distributed Systems**, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[Krol91] Krol, Th., **A Generalization of Fault-Tolerance Based on Masking**, Ph.D. thesis, Eindhoven University of Technology, 1991.

[LaMJ91] Laranjeiri, L., Malek, M., and Jenevein, R., On tolerating faults in naturally redundant algorithms, in: **Proceedings of the 10th Symposium on Reliable Distributed Systems, Pisa, Italy, September 1991**, IEEE Computer Society Press, Los Alamitos (CA), 1991, pp.118-127.

[Lapr85] Laprie, J.C., Dependable Computing and Fault-Tolerance: Concepts and Terminology, in: **Proceedings of the 15th International Symposium on Fault-Tolerant Computing Systems (FTCS-15)**, Ann Arbor, Michigan, June 1985, pp. 2-11.

[Lapr92] Laprie, J.C., **Dependability: Basic Concepts and Terminology - In English, French, German, and Japanese**, Springer-Verlag, Vienna, 1992.

[Lapr94] Laprie, J.C., Dependability: The Challenge for the Future of Computing and Communication Technologies, in: **Dependable Computing - EDCC-1, First European Dependable Computing Conference, Berlin, Germany, October 1994, Proceedings**, Echtle, K., Hammer, D., and Powell, D. (Eds.), Springer-Verlag, LNCS 852, 1994, pp. 407-408.

[Lapr95] Laprie, J.C., Dependability - Its Attributes, Impairments and Means, in: **Predictably Dependable Computing Systems**, Randell, B. et al. (Eds.), Esprit Basic Research Series, Springer-Verlag, 1995, pp.3-24.

[LaSP82] Lamport, L., Shostak, R., and Pease, M., The Byzantine Generals Problem, in: **ACM Transactions on Programming Languages and Systems**, Vol. 4, No. 3, July 1982, pp.382-401.

[MaSl78] MacWilliams, F.J., and Sloane, N.J.A., **The Theory of Error-Correcting Codes**, Amsterdam, the Netherlands, North-Holland, 1978.

[Powe91] Powell, D. (Ed.), **Delta-4: A Generic Architecture for Dependable Distributed Computing**, Research Reports ESPRIT, Project 818 / 2252, Delta-4, Vol.1, Springer-Verlag, ISBN 3-540-54985-4, 1991.

[ScSc83] Schlichting, R., and Schneider, F.B., Fail-stop processors: an approach to designing fault-tolerant computing systems, in: **ACM Transactions on Computer Systems**, Vol.1, No.3, August 1983, pp.222-238.

[SiSw92] Siewiorek, D.P., and Swarz, R.S., **RELIABLE COMPUTER SYSTEMS: DESIGN AND EVALUATION**, 2nd edition, Digital Press, Bedford (MA), ISBN 1-555-58075-0, 1992.

[Triv82] Trivedi, K.S., **Probability and Statistics with Reliability, Queueing, and Computer Science Applications**, Englewood Cliffs, New Jersey, Prentice Hall, 1982.

[VaOo89] Vanstone, S.A., and van Oorschot, P.C., **An Introduction to Error Correcting Codes with Applications**, Kluwer Academic Publishers, Boston, 1989.

CHAPTER 2

A System Model

"It did not take long before the scientists began to notice that complex systems showed certain common behaviors. They started to think of these behaviors as characteristic of all complex systems. They realized that these behaviors could not be explained by analyzing the components of the systems."

Michael Crichton, The lost world

Abstract

In this chapter, a system model will be defined, as it is used in the following chapters of this thesis. Furthermore, as a motivation for later chapters, a system for dependable data storage based on masking redundancy in combination with recovery and diagnosis algorithms, is proposed. The algorithms described in later chapters can conveniently be used in this so-called Dependable Distributed Data Storage (DDDS) system. Finally, it will be argued why resilience to Byzantine faults in the DDDS system and in the protocols in later chapters is desirable.

2.1. Introduction

In Chapter 1, it is argued that a repairable system in which a high reliability or availability is required, should be based on *masking redundancy*. In practice, in order to make the system resilient not only to transient and intermittent faults, but to permanent faults as well, masking redundancy will always be implemented using *hardware* techniques.

One of the goals of our research is to develop a number of fault-tolerant protocols that can be used in a highly reliable system. In order to make these protocols resilient to a number of arbitrary processor failures, hardware masking redundancy techniques should be applied on system level in the system on which the protocols are executed, since hardware redundancy applied on system level provides resilience to permanent processor faults. In a system based on system-level hardware masking redundancy, the system data and the services provided by the system are physically distributed among a number of processors. The underlying system model, as presented in Section 2.2. will be that of a ***distributed system***.

As an introduction to the subsequent chapters, in which a number of Byzantine fault-tolerant protocols for a distributed system will be presented, in Section 2.3., a description of the Dependable Distributed Data Storage (DDDS) system is given. This is a virtual system for reliable distributed data storage, which consists of a set of processors, a number of which may be arbitrarily faulty. The purpose of the description of this system is to illustrate how the fault-tolerant protocols presented in Chapters 3, 4, and 5 may be applied. However, it is important to keep in mind that neither the DDDS system is meant to be the optimal system for reliable data storage, nor is application of the protocols in Chapter 3, 4, and 5 restricted to the DDDS system. The proposed protocols, especially those presented in Chapter 3 and 5, are applicable in almost every area of fault-tolerant distributed computing.

2.2. System model

In this section, the system model is introduced. This system model provides a specification of the systems in which the protocols described in Chapter 3, 4, and 5 can be used. Before the system model can be defined, first, the definition of a system should be given.

In [AnLe81], a ***system*** is defined as an identifiable mechanism that maintains a pattern of behaviour at an interface between the system and its environment. This definition of a system is from the point of view of its external behaviour. A ***system specification*** usually aims at prescribing the external behaviour which the system should exhibit. The behaviour of the system is decided to be correct when the external behaviour of the system satisfies the system specification.

In general, a system is considered as being composed of a number of components or ***subsystems***, which interact according to the design of the system. Each of the subsystems can, in its turn, be regarded as a system in its own right, with its own internal structure and its own external behaviour. In the same way, any distinguished subsystem can be considered as being composed of a number of components at a lower level. This hierarchical system / subsystem decomposition continues until a level beyond which it is impossible or undesirable to further specify the details of the system. In literature, the decomposition usually stops at the level of components or gates, or at the level of processors. The latter level is often called the ***system level***.

As stated in Section 1.1.4., by applying *hardware* redundancy in a system, the system can be made resilient to a number of *permanent* faults[1] of the components of the system. By applying redundancy on system level (i.e., the system consists of a number of processors that perform the same task), resilience to a number of permanent *processor* failures can be obtained.

The underlying physical model of the system is that of a so-called ***distributed system***. In [Jalo94, p.46], a distributed system is defined as a system consisting of a number of processors (sometimes also called ***nodes***), that are geographically at different locations, and connected by a communication network. The processors communicate with each other by exchanging messages over the communication network.

A key property of a distributed system is the ***geographical separation*** of the various processors in the system, i.e., the processors are physically placed at different loca-

1. Resilience against permanent faults is obtained in addition to resilience to transient and intermittent faults, which may also be obtained using software redundancy. See also Section 1.1.4.

tions.

Another important feature of a distributed system is the ***absence of shared memory*** among different processors, i.e., there is no memory in the system that is accessible to more than one processor. In contrast, in ***parallel systems***, different processors may contain shared memory, which they usually apply to communicate with each other.

A final feature of a distributed system is the ***absence of a global clock***, i.e. every processor has its own processor clock. Again, this is in contrast with parallel systems, in which it is common to have one global clock drive all processors in the system.

The last two properties, i.e., absence of shared memory and absence of a shared clock, are often summarized by saying that all processors in a distributed system are ***autonomous*** (or ***loosely coupled***). This is in contrast with ***parallel systems***, where processors are ***closely coupled***, i.e., they are not autonomous [Jalo94, p.46].

As mentioned before, the processors in a distributed system communicate with each other by exchanging messages over the communication network. The communication network consists of a number of so-called ***communication links***. In a so-called ***point-to-point network***, each communication link directly connects two processors in the system. Other types of networks exist, e.g., ***bus networks*** or ***ring networks***, or networks in which processors are connected via zero or more intermediate relay processors. See Figure 2.1. The type of network that is required is protocol-dependent. E.g., the protocol described in Section 3.2. requires a point-to-point network, whereas the protocols presented in Section 3.3., 3.4., and Chapter 4 and 5 may also be applicable in the other types of networks mentioned above[2].

The way in which the nodes are connected in the communication network is called the ***network topology***. For point-to-point networks, various network topologies are possible. Some common network topologies are the star, the tree, or the fully-connected topology. See Figure 2.1.

In all protocols described in the following chapters, it is assumed that the system consists of a set *N*, of *N* processors, up to *T* of which may be faulty. Here, the variable *T* is a design parameter, which indicates the maximum number of faulty processors that is allowed. Implicitly, the protocols are based on the assumption that the higher the value of *T* for a given *N*, the more reliable the system will be. In other words, we assume that processors fail independently. This assumption, which is, actually, often made in the field of reliable computing, is justified by the fact that the processors in the distributed system are autonomous and geographically distributed.

Notice that the design parameter *T* determines the reliability of the system, i.e., the reliability is expressed as the maximum number of processors that may simultaneously behave maliciously. We are aware of the fact that there exist other ways to express the reliability of a system, and to show that a certain reliability improvement has been

2. In fact, these protocols require the logical communication network to be fully-connected, and they are based on the assumption that link failures do not occur. In a point-to-point network, this assumption is justified by the fact that a link failure can be modeled as a failure of one of its adjacent processors. In other networks, this is only the case if links fail independently.

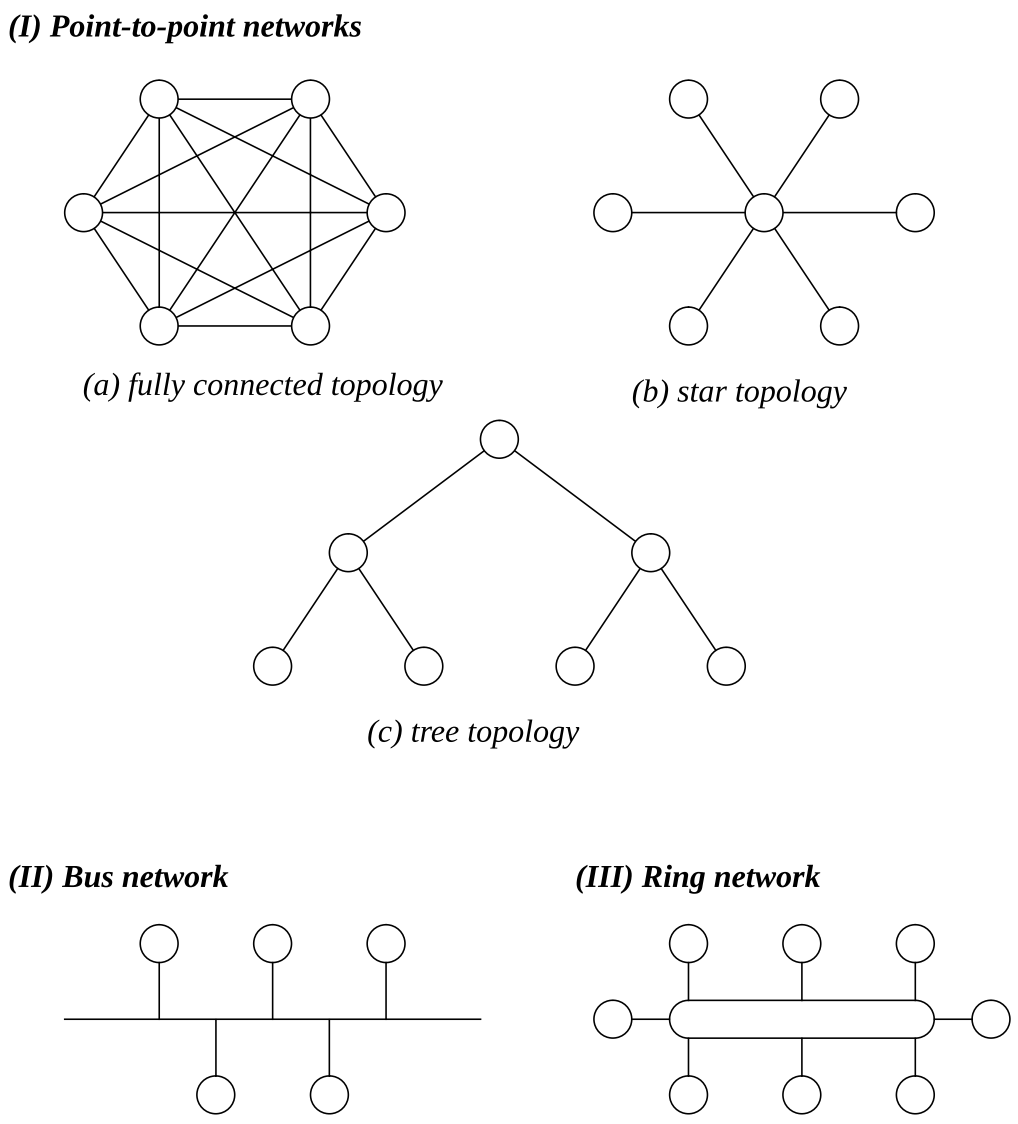


Figure 2.1. Different types of communication networks

achieved. Most of these methods, however, yield results that are highly dependent on the technical quality of the components of the distributed system under consideration. In practice, such methods require to be continuously revised, since the technical quality of system components improves continuously. Notice that the method that we have selected allows mathematical proofs that the claimed reliability improvement (in terms of the maximal number of processors that may simultaneously behave maliciously) is achieved, independent of the technical quality of the components of the system under consideration.

2.3. Description of the Dependable Distributed Data Storage system

In this section, we will introduce the so-called Dependable Distributed Data Storage (DDDS) system. The DDDS system has been described in detail in [Boer96, Hart95, Wess95] [3]. It is based on masking redundancy techniques combined with diagnosis and recovery algorithms, and aims at providing an opportunity to store data in a reliable way. The system can be built from off-the-shelf hardware components. It consists of a set, *N*, of *N* processors, and provides resilience to arbitrary (or Byzantine) failures of up to *T* of its processors. The DDDS system enables several users to store their files in a reliable way.

Users that want to store data in a reliable way in the DDDS system have to become a ***member*** of the system. The DDDS system has a variable number of members. The data of the members of the DDDS system is distributed across the disks of the processors that are part of the DDDS system. When becoming a member of the DDDS system, a user makes some disk space of his / her processor available for reliable data storage. The user's processor has now become part of the DDDS system. Since the number of processors in the DDDS system has now increased, it may be necessary to re-distribute the data of the members of the DDDS system across the disks of the processors that are part of the DDDS system. The system structure is invisible to the members of the DDDS system, and they do not need to bother where their files are stored in the system. A schematic illustration of the DDDS system is given in Figure 2.2. In this

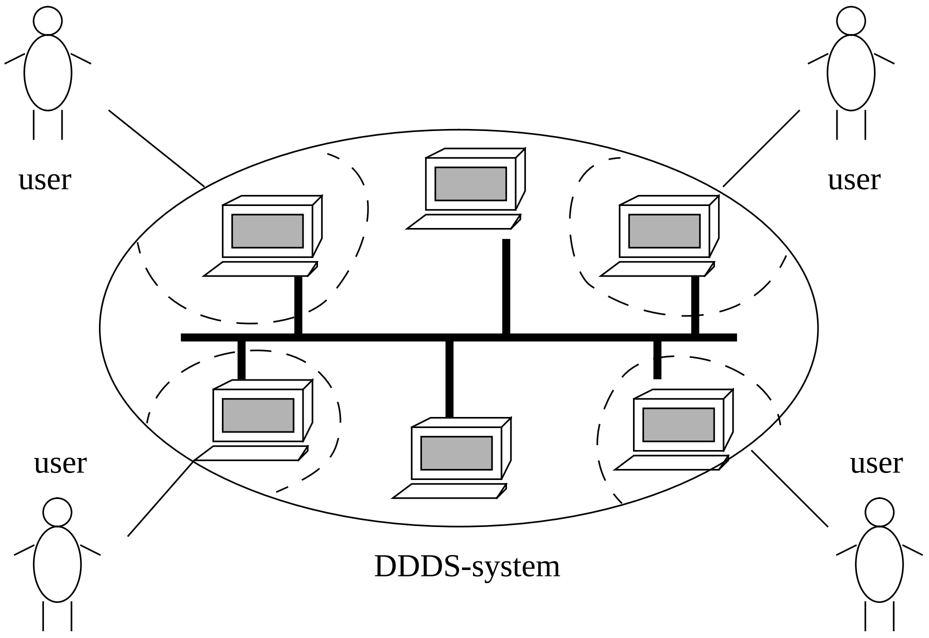


Figure 2.2. Schematic illustration of the DDDS system

illustration, a bus network connects the various processors. In an actual implementation of the DDDS system, other types of networks are also possible. See Section 2.2.

An important feature of the DDDS system is that ***single points of failure do not occur***,

3. In these publications, the system has been referred to as Scattered Backup System (or SBS), since backups of files are scattered among the processors in the system. However, the system is not intended to serve as a backup system. Therefore, in order to prevent confusion, we rather use the name Dependable Distributed Data Storage system, since the described system is a data storage system rather than a backup system.

except for the fact that users cannot be prevented from corrupting or deleting their own files. So, in a way, a user can be viewed as a single point of failure for updates on his / her own files. This problem can be circumvented by not allowing deletion of files, and creating and storing a new version of a file at every update. In this way, the amount of data stored in the system continuously increases. However, in any practical implementation of the DDDS system, there will always exist limits on the amount of data that can be stored in the system, so, either, we have to regularly backup old data on a tape (after which the old data can be removed from the system), or concessions have to be made, and wise management is required in order to decide which files should be removed and which not. This so-called ***version management*** is beyond the scope of this thesis, however.

The main functionality of the DDDS system is to provide reliable storage of data. This functionality is delivered by a number of so-called ***replicated system services***. A replicated system service is a replicated process, executed on several processors in the system, which delivers the requested service. Note that the system services are necessarily replicated in order to avoid a single point of failure. (Viz., if the system service would be provided by a process running on a single processor, then failure of this single processor would cause a system breakdown.)

Any replicated system service in the DDDS system consists of a number of identical copies (also called ***replicas***) of a process, each of which is executed on a different processor. The members of the DDDS system may request service from the system by sending their request to every processor on which a copy of the service process is executed. The output of two process replicas executed on different correctly functioning processors can only be guaranteed to be equal, provided that the same request has been received by both replicas. However, a maliciously behaving member may send different requests to different replicas, as a result of which the output of any two correct replicas may be different. So, in order to avoid inconsistency between correct replicas of a system service, it must be ensured that the correct replicas agree on any information sent to them by members of the system. A solution to this so-called ***agreement problem*** is to use the authenticated self-synchronizing Byzantine Agreement Protocols introduced in Chapter 3. These protocols, which are applicable in a distributed system, guarantee that all correct replicas agree on information sent to them by any arbitrary member, provided that the number of maliciously behaving processors in the system stays below a certain predefined maximum.

In the DDDS system, the following replicated system services can be distinguished:
❑ data storage and retrieval service
❑ data recovery service
❑ diagnosis service
❑ membership management service
❑ version management service

Each of these services will be briefly investigated below.

The ***data storage and retrieval service*** is responsible for reliable storage of data, and for retrieval of data stored in the system. In the DDDS system, the files are stored in a fault-tolerant way on a set of disks located in different processors. This is done by splitting the file into a number of file fragments. A number of redundant fragments is constructed such that the file fragments and the redundant fragments form code words of an erasure-correcting code. Then, the fragments are distributed among the disks of a number of processors. The members of the DDDS system will encrypt their files in order to protect them against unauthorized reading and undetected mutilation by other members of the system[4]. The data of the *replicated system services* should also be encrypted in order to protect it against unauthorized reading and undetected mutilation by the members of the system. Storage of the data on a set of disks of different processors is the responsibility of the data storage and retrieval service. The members of the DDDS system need only put their files at the disposal of the data storage and retrieval service. This is established by sending the file fragments to every processor on which a replica of the data storage and retrieval service resides. The members may retrieve one or more of their files by sending a request to all replicas of the data storage and retrieval service, as a result of which they obtain the file fragments of the requested files.

Since the file fragments of any file of any member are stored on disks of a number of processors, failure of one or more of these processors may lead to loss or corruption of a number of file fragments. Although sufficient redundant file fragments may be available in order to enable reconstruction of the original file, it is still important to recover the lost or corrupted file fragments as soon as possible, because only after the lost or corrupted file fragments have been recovered, the system is able to tolerate subsequent processor failures. However, if the system performs a critical task at the moment that a processor failure occurs, it may be desirable to postpone recovery of the lost or corrupted file fragments till the moment at which the system has finished its critical task.

The responsibility for recovery of lost or corrupted file fragments is given to the ***data recovery service***. Notice that the responsibility for recovery of lost or corrupted file fragments can *not* be given to a single processor. Whether or not recovery can take place would then depend on the correctness of a single processor, hence, the system would then contain a single point of failure. How the data recovery service can be implemented will extensively be discussed in Chapter 4.

The ***diagnosis service*** is responsible for localization of failed processors in the system. It is very important that the results of the performed diagnoses are correct, since all processors diagnosed as being faulty will be excluded from the system in the reconfiguration process that follows every diagnosis. The correctness of the diagnoses depends on the quality of the testing procedures used to make up the diagnoses, and of the fact whether or not the diagnosis process is correctly executed. The latter depends on the correctness of the processors on which the replicas of the diagnosis process are executed. Since the diagnosis process is a replicated process, a number of processor failures in the processors on which the replicas reside, can be masked. However, in order to be able to guarantee optimal resilience to subsequent processor failures, it is of key importance to exclude the faulty processors from the diagnosis service itself as soon as possible. The latter implies that the diagnosis service should (at least periodically) test the processors on which the service itself resides! Designing good testing algorithms is

4. Encryption of file fragments is necessary, since the file fragments of the files of one member may be stored on the disks of other members of the DDDS system, a number of which may behave maliciously.

a nontrivial task, which is beyond the scope of this thesis. Tests which check if a processor is correct usually check, for a limited number of test cases, whether the behaviour of a processor corresponds to what was expected. In general, it is cumbersome or even impossible to have the test cases cover every situation that may occur. As a consequence, most practical tests have an ***imperfect fault coverage***, i.e., execution of a test to find all faulty processors cannot guarantee that all faulty processors will indeed be found.

The diagnosis service can be implemented by using the new diagnosis algorithm introduced in Chapter 5. This algorithm enables removal of all detectably faulty processors from a fault-tolerant service, under the assumption that the tests may have an imperfect fault coverage. It is assumed that, prior to execution of this diagnosis algorithm, the processors have tested each other, and that all correct processors have reached Byzantine Agreement about the test results. The latter can be guaranteed by having all correct processors distribute their test results to all other processor by means of execution of an authenticated self-synchronizing Byzantine Agreement Protocol (these protocols will be introduced in Section 3.3.). Execution of the diagnosis algorithm on a set, N, of N processors, up to T of which may behave maliciously, results in agreement among all correct processors about the group of all detectably faulty processors in N.

The ***membership management service*** allows users to become a member of the DDDS system, and allows members of the system to cancel their membership. When becoming a member of the DDDS system, a user makes an amount of disk space on the disk of his / her own processor available for reliable data storage. In return, the user gets permission to reliably store a certain amount of data. If a member cancels its membership of the DDDS system, then all file fragments of other members stored on its disk are moved to other processors in the system, after which the file fragments of the departing member are returned. A more detailed investigation of the membership management service can be found in [Hart95].

As stated in the beginning of this section, the ***version management service*** is responsible for deciding which versions of the files may be deleted and which files must be kept stored in the system. Elaboration of the version management service is beyond the scope of this thesis.

2.4. Resilience to Byzantine faults

As stated in the previous section, one of the features of the DDDS system (and of the protocols which will be described in Chapter 3, 4, and 5) is the resilience to a (limited) number of Byzantine faults. In Section 1.1.2.2., the class of Byzantine faults is defined as the universal fault class, i.e., containing every possible fault. Since, in general, it is impossible to make a system resilient to any type of fault, the class of Byzantine faults (which should be tolerated) is usually (explicitly or implicitly) restricted to the class of all possible faults that are relevant for the considered protocols in the system model under consideration. In order to avoid confusion, it is better to speak of a ***restricted Byzantine fault model***.

This section starts with the presentation of a restricted Byzantine fault model, which is often assumed in literature on Byzantine fault-tolerant protocols for so-called ***lock-step***

synchronous systems. An exact definition of a lock-step synchronous system will be given in Section 3.1.6. Here, it suffices to say that, in fact, in such a system, a global clock is assumed to drive all processors in the system. All correct processors in the system are assumed to know the start of every distributed protocol that is executed by them, and to execute every step in each protocol simultaneously.

As shown below, the restricted Byzantine fault model for lock-step synchronous systems turns out to be too restrictive for Byzantine fault-tolerant protocols in a distributed system. Therefore, we will present a new, broader, restricted Byzantine fault model, which contains all faults that are relevant for Byzantine fault-tolerant protocols in a distributed system.

One may wonder whether it is realistic to build a system that is resilient to *Byzantine* faults (within some restricted Byzantine fault model). Below, we will give our motivation for designing a system that is resilient to a limited number of Byzantine faults.

It is a well-known fact that tolerating Byzantine faults is expensive, in software, as well as in computation time and in hardware. Cristian et al. (in [CASD95]) show that, if compared to timing and omission faults, the cost of tolerating Byzantine faults in fault-tolerant protocols is high. In the Delta-4 ESPRIT-project [Powe91] this has been the motivation to restrict the system fault-tolerance to timing and omission faults [Powe91, p.274].

However, it is also a fact that in every computer system, faults may occur, that do not fall into any of the fault classes distinguished in Section 1.1.2.2., other than that of the Byzantine faults. In order be able to tolerate these faults, the system should at least be made resilient to those Byzantine faults which are defined by the appropriate restricted Byzantine fault model. For several critical applications, the probability of system failure should be made as small as possible. For these applications, the cost of a system breakdown may easily outweigh the cost of tolerating Byzantine faults. Since the focus of this thesis is on the design of protocols for systems in which a high reliability and availability is required, this justifies the high costs involved in tolerating Byzantine faults.

2.4.1. A restricted Byzantine fault model for lock-step synchronous systems

We will now first present the restricted Byzantine fault model, as it is often assumed in literature on Byzantine fault-tolerant protocols for lock-step synchronous systems. This restricted Byzantine fault model has, e.g., been used in [LaSP82, Krol91].

In [Krol91], Byzantine faults are regarded as equivalent to faults causing ***broadcast errors***. Broadcast errors are defined in the context of a fault-tolerant system based on system-level masking redundancy in a *lock-step synchronous system*. In such a fault-tolerant system, a single processor (called source processor) communicates with a fault-tolerant service by sending identical data to all processors on which a replica of the fault-tolerant service resides. The source processor is said to exhibit a ***broadcast error***, if it sends different data to different replicas. In the restricted Byzantine fault model in [Krol91], a Byzantine fault is equivalent to a fault which causes a processor to exhibit a broadcast error, i.e. to send different data to different processors. Notice that this type of fault is not covered by any fault class distinguished in Section 1.1.2.2. other than that of Byzantine faults. The class of Byzantine faults defined by the restricted Byzantine fault model in [Krol91] and described above is equivalent to that of the ***malicious-asymmetric faults*** described in [SuHW94]. In [Lapr95, Powe91], this fault class is referred to as the class of faults liable to lead to ***value failures*** (See Figure 1.1.).

Clearly, the above restricted Byzantine fault model does not cover all possible faults. However, the fault model is defined such that it covers all possible faults that are relevant for the protocols under consideration, executed in a lock-step synchronous system.

In [Krol91], Krol emphasizes the relevance of being able to tolerate the Byzantine faults specified in the above restricted Byzantine fault model by showing that broadcast errors may occur in any system based on masking redundancy. In general, the logic discrimination levels as well as any two sampling instants of two different processors in the system will always be different. Broadcast errors are typically caused by source processors multicasting ambiguous logic levels (see Figure 2.3.) or by timing faults (see Figure 2.4.).

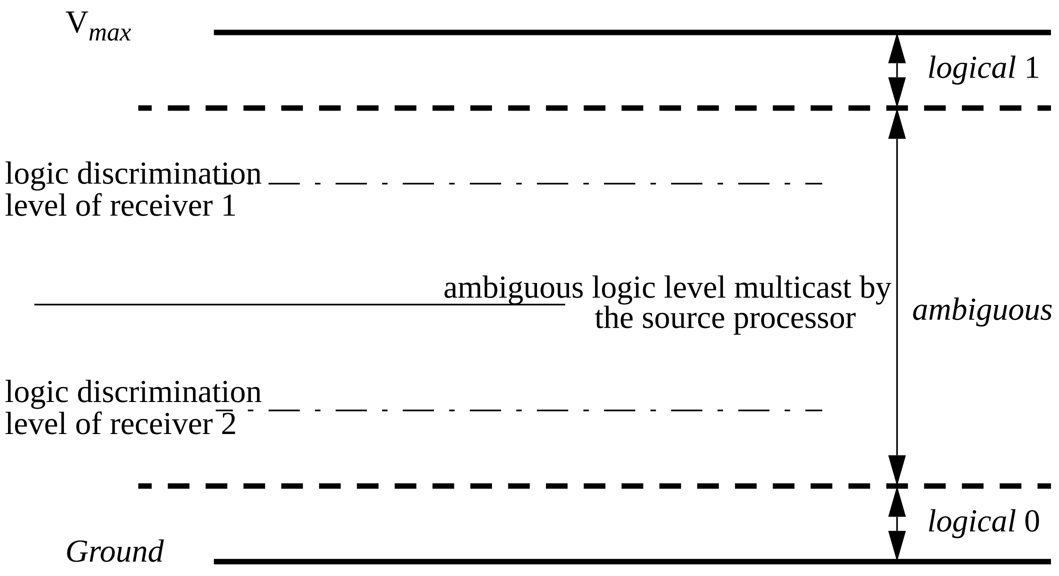

V_max
logical 1
logic discrimination level of receiver 1
ambiguous logic level multicast by the source processor
ambiguous
logic discrimination level of receiver 2
logical 0
Ground

Figure 2.3. Broadcast error due to a source processor multicasting an ambiguous logic level

In Figure 2.3., a broadcast error is caused by the fact that the source processor multicasts an ambiguous logic level. The difference in the logic discrimination levels of receiver 1 and 2 causes receiver 1 and 2 to arrive at different decisions about the received signal: receiver 1 will decide that a logical 0 has been received, while receiver 2 decides a logical 1.

In Figure 2.4., a broadcast error is caused by a timing fault in the source processor in combination with different sampling instants of receivers 1 and 2. Due to a timing fault

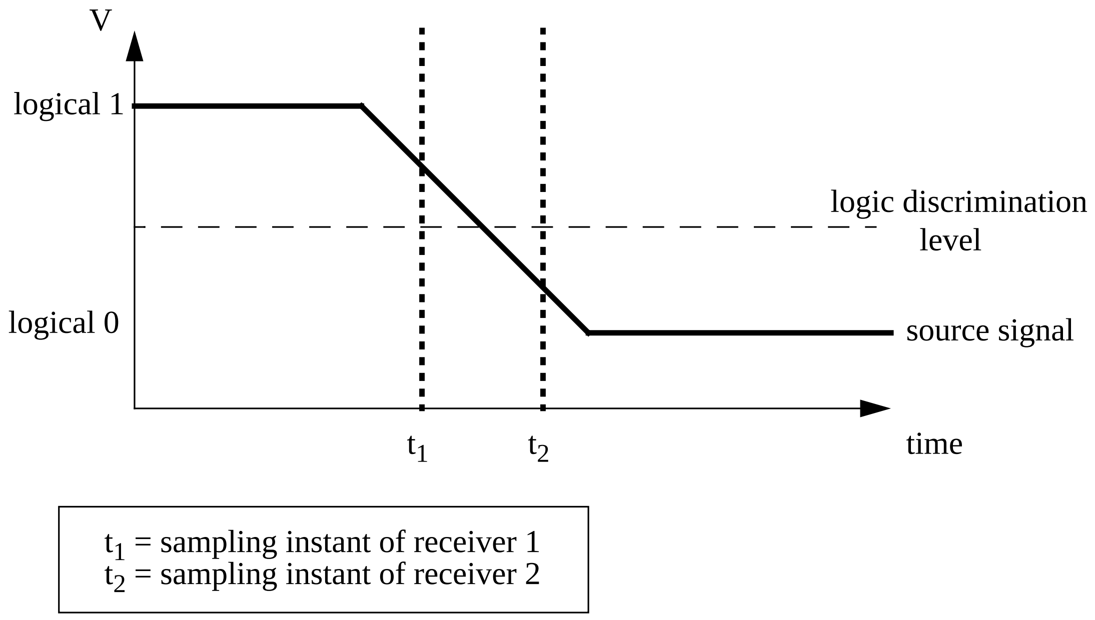

V
logical 1
logical 0
logic discrimination level
source signal
time
t_1
t_2
t_1 = sampling instant of receiver 1
t_2 = sampling instant of receiver 2

Figure 2.4. Broadcast error due to a timing fault in the source processor and different sampling instants of receivers 1 and 2

of the source processor, both receivers will take a sample of the source signal when it changes from logical 1 to logical 0. The difference in sampling instants causes receiver 1 and 2 to arrive at different decisions about the value of the source signal: receiver 1 will decide a logical 1, while receiver 2 decides a logical 0.

2.4.2. A restricted Byzantine fault model for a distributed system in which protection against malicious intruders is required

In this section, we will extend the restricted Byzantine fault class introduced in the previous section in order to make it cover all relevant Byzantine faults that may occur in a distributed system in which protection against malicious intruders is required. It will be argued that, besides the faults contained in the restricted Byzantine fault class of Section 2.4.1., resilience is needed against timing faults and collusion.

In Section 1.1.2.2., the class of ***timing faults*** has been defined as a fault class encompassing any fault that occurs when a processor completes its task either before or after its specified time frame or never.

In a *lock-step synchronous system,* it is impossible for a processor to complete its task before or after its specified time frame. In such a system, either a processor completes its task in time, or it never completes its task. In other words, in a lock-step synchronous system, the class of timing faults is equivalent to the class of omission faults.

In a *distributed system*, however, timing faults may cause processors in the system to complete their tasks before or after a specified time frame. Since timing faults form a subclass of the class of Byzantine faults (see Figure 1.2.), timing faults have to be taken into account in the definition of the restricted Byzantine fault model. This can be accomplished by slightly changing the definition of a broadcast error.

We will now give a definition of broadcast errors in the context of a fault-tolerant system based on system-level masking redundancy in a *distributed system*. In such a system, a single processor (called source processor) communicates with a fault-tolerant service by sending identical data to all processors on which a replica of the fault-tolerant service resides. The source processor is said to exhibit a ***broadcast error***, if it sends different data to different replicas *at arbitrary times* (i.e., within or outside the specified time frame).

In the restricted Byzantine fault model for a distributed system, a Byzantine fault is equivalent to a fault which causes a processor to exhibit a broadcast error, i.e., to send different data to different processors at arbitrary times.

Of course, in systems, in which one has to cope with malicious intruders trying to break the system, the possibility for Byzantine faults to occur is evident. It should be taken into account that an intruder may try everything to cause a system breakdown. It is important to realize that such attacks can also be attempted from *inside* the system, once an intruder has managed to compromise one or more of the processors in the system.

A possible approach to cope with such attacks is to apply cryptographic techniques in the system. Cryptography provides a solution to problems with regard to privacy and authentication of information. Solutions to the ***privacy*** problem aim to inhibit extraction of information by unauthorized parties from a certain piece of data communicated or stored in the system. Solutions to the ***authentication*** problem try to guarantee a possibility to verify the identity of the creator of a certain piece of data communicated or stored in a system, provided that the data is uncorrupted. Furthermore, authentication provides a means for ***mutilation detection***, i.e., authentication offers a possibility to detect mutilations of the data by unauthorized third parties.

An additional advantage of applying authentication is that the class of faults that should be tolerated can be restricted to that of the ***authenticated Byzantine faults***. The possibility to detect mutilations of authenticated information allows the use of so-called ***authenticated protocols*** that, in order to provide resilience to a certain maximum number of faulty processors in the system, require less communication overhead and fewer redundant processors in the system than protocols not using authentication. In an authenticated protocol, every correct processor authenticates every message that it communicates to other processors with a secret cryptographic key which is unknown to other processors and which they are not able to obtain. The other processors are, however, able to verify the correctness of the message. Usually, a number of assumptions is made with regard to the authentication of messages. E.g., in [LaSP82], it is assumed that a correct processor's signature cannot be forged and that any alteration of its signed messages can be detected. No assumptions are made about a faulty processor's signature. In particular, ***collusion*** among faulty processors is allowed, i.e., a faulty processor's signature may be forged by other faulty processors.

Throughout this thesis, this restricted authenticated Byzantine fault model has (implicitly) been used. The protocols introduced in Chapter 3, 4, and 5 provide resilience to a limited number of faults within the restricted authenticated Byzantine fault model defined above. If, in the sequel, a processor is said to be (arbitrarily) faulty, or, equivalently, to behave maliciously, it is assumed that this processor may exhibit faults within the above-defined restricted authenticated Byzantine fault model.

In [Lapr94], Laprie emphasizes the importance of tolerating such authenticated Byzantine faults by stating that “it is estimated that currently half of the total cost of computer failures is due to malicious faults (thus related to security issues) and this cost proportion has continuously increased for ten years”.

2.5. Conclusion

In this chapter, a system model has been introduced. This system model specifies the systems in which the protocols, which will be introduced in subsequent chapters, can be implemented. As an introduction to the protocols in Chapter 3, 4, and 5, a description of the Dependable Distributed Data Storage (DDDS) system is given. The DDDS system is a virtual system based on hardware masking redundancy techniques in combination with recovery and diagnosis algorithms, which aims at providing reliable data storage. It is clearly indicated how the protocols of Chapter 3, 4, and 5, can be implemented in the DDDS system. Finally, it is argued why resilience to Byzantine faults is desirable for the DDDS system and for the protocols in Chapter 3, 4, and 5.

2.6. References

[AnLe81] Anderson, T.A., and Lee, P.A., **Fault Tolerance Principles and Practice**, Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[Boer96] Boer, W. de, **Applying Cryptography in a Scattered Backup System**, Graduation thesis SPA-96-06, University of Twente, Enschede, May 1996.

[CASD95] Cristian, F., Aghili, H., Strong, R., and Dolev, D., Atomic broadcast: From simple message diffusion to Byzantine Agreement, in: **Information and Computation**, Vol 118, 1995, pp.158-179. (An earlier version of this paper appeared in: **Proceedings of the 15th International Symposium on Fault-Tolerant Computing**, Ann Arbor, June 1985, pp.200-206)

[Hart95] Hartman, G., **Membership Management of the Scattered Backup System**, Graduation thesis SPA-95-16, University of Twente, Enschede, October 1995.

[Jalo94] Jalote, P., **Fault Tolerance in Distributed Systems**, Prentice Hall, Englewood Cliffs, New Jersey, 1994.

[Krol91] Krol, Th., **A Generalization of Fault-Tolerance Based on Masking**, Ph.D. thesis, Eindhoven University of Technology, 1991.

[Lapr94] Laprie, J.C., Dependability: The Challenge for the Future of Computing and Communication Technologies, in: **Dependable Computing - EDCC-1, First European Dependable Computing Conference, Berlin, Germany, October 1994, Proceedings**, Echtle, K., Hammer, D., and Powell, D., (Eds.), Springer-Verlag, LNCS 852, 1994, pp. 407-408.

[Lapr95] Laprie, J.C., Dependability - Its Attributes, Impairments and Means, in: **Predictably Dependable Computing Systems**, Randell, B., et al. (Eds.), Esprit Basic Research Series, Springer-Verlag, 1995, pp.3-24.

[LaSP82] Lamport, L., Shostak, R., and Pease, M., The Byzantine Generals Problem, in: **ACM Transactions on Programming Languages and Systems**, Vol. 4, No. 3, July 1982, pp.382-401.

[Powe91] Powell, D. (Ed.), **Delta-4: A Generic Architecture for Dependable Distributed Computing**, Research Reports ESPRIT, Project 818 / 2252, Delta-4, Vol.1, Springer-Verlag, ISBN 3-540-54985-4, 1991.

[SuHW94] Suri, N., Hugue, M.M., and Walter, C.J., Synchronization Issues in Real-Time Systems, in: **Proceedings of the IEEE**, Vol. 82, No. 1, January 1994, pp.41-54.

[Wess95] Wessels, J., **Initial Analysis and Modelling of the Scattered Backup System**, Graduation thesis SPA-95-01, University of Twente, Enschede, January 1995.

CHAPTER 3

Authenticated Byzantine Agreement Protocols

"De aktiviteiten welke wij ontwikkelen om gegevens te verzamelen, zullen op minstens even grote schaal door een tegenstander worden ontwikkeld. Om een tegenstander dus niet in een voordelige positie te brengen, waardoor de eigen eenheid onherstelbare schade kan lijden, zal een aantal maatregelen moeten worden getroffen."

HANDBOEK VOOR DE SOLDAAT, VS 2-1350,
KONINKLIJKE LANDMACHT

Abstract

In order to make a dependable distributed computer system resilient to arbitrary failures of its processors, deterministic interactive consistency algorithms (ICAs) can be applied. The design of a deterministic interactive consistency algorithm boils down to the design of a deterministic Byzantine Agreement Protocol (BAP).

The first deterministic Byzantine Agreement Protocols were described by Lamport et al. These BAPs are, however, impractical for two reasons. First, the BAPs require a large communication overhead, and second, application of these BAPs is confined to lock-step synchronous systems in which all correct processors know the start of the BAP and start the BAP simultaneously. A vast amount of literature has appeared on BAPs, which require less communication overhead than the BAPs described by Lamport et al. Only very few attempts have been undertaken in order to make deterministic BAPs also work in a system with less strict synchronicity requirements. In this chapter, both problems will be addressed: first, an alternative way to reduce the required communication overhead in a BAP will be considered; after that, we introduce a new class of BAPs, that work in a system with less strict synchronicity requirements.

First, we describe a new class of authenticated BAPs for lock-step synchronous systems, in which the required communication overhead is decreased by reducing the size of the messages. This is accomplished by encoding messages into symbols of an erasure-correcting code instead of multicasting these messages as is done in the original authenticated BAP described by Lamport et al. In a number of relevant cases, our new authenticated BAPs require considerably less data communication than other authenticated BAPs in literature, whereas the required number of processors and communication phases meet the minimum bounds. Our new authenticated BAPs are defined and proved on basis of a class of algorithms called the Authenticated Dispersed Joined Communication Algorithms.

Subsequently, we investigate a new class of so-called authenticated self-synchronizing BAPs which work in a system with less strict synchronicity requirements.

Many BAPs found in literature require that communication takes place in synchronous rounds of information exchange and require that all correct processors know the start of the BAP and start the protocol simultaneously. Because it is hard to satisfy either or both requirements in a distributed system, several authenticated BAPs have been proposed in which the processors need not know the start of the protocol. These BAPs only require the processor clocks of all correct processors to be approximately synchronized at the start and during execution of the BAP. However, this requires the system to contain at least a majority of correctly functioning processors, which periodically execute a deterministic fault-tolerant clock synchronization algorithm. In this chapter, we define authenticated self-synchronizing BAPs, which overcome these restrictions by guaranteeing Byzantine Agreement for any number of faulty processors in the system and arbitrary clock skew between the clocks of the processors. In our authenticated self-synchronizing BAPs, the processors need not know the start of the BAP. A disadvantage of these self-synchronizing BAPs is that they are inefficient with regard to the number of messages needed. Therefore, we also define a class of optimized authenticated self-synchronizing BAPs, that require less communication overhead than the original authenticated self-synchronizing BAPs, be it at the cost of increased lengths of the communication phases of the protocol, if compared to the original authenticated self-synchronizing BAPs. Which protocols are to be preferred depends on the implementation that is considered.

Finally, we describe the design of an authenticated self-synchronizing ICA respectively an optimized authenticated self-synchronizing ICA, which is based on the design of an authenticated self-synchronizing BAP respectively an optimized authenticated self-synchronizing BAP.

3.1. Introduction

A dependable distributed computer system can withstand the failure of one or more of its processors, if every system service is provided by a set of replicated tasks running on different processors. However, malfunctioning client machines communicating with such a fault-tolerant system service may produce broadcast errors (i.e., send conflicting information to the different replicas of the system service) as a result of which inconsistency between the replicas of the system service may occur. This may lead to a system breakdown, even if the system does not contain more faulty processors than it is designed to tolerate[1].

In order to avoid inconsistency between the correct replicas of the system service, it must be ensured that the correct replicas agree on any information sent to them by clients. In literature, many solutions to this so-called ***agreement problem*** have appeared.

Roughly, the proposed solutions can be divided into two different groups: ***consensus protocols*** and ***interactive consistency algorithms*** respectively, which are defined in Section 3.1.1. and 3.1.2. respectively. Consensus protocols and interactive consistency

1. This problem is called the ***Input Problem***. See [Krol91] for details.

algorithms may solve the agreement problem even in the presence of arbitrarily faulty (or ***malicious***) processors.

As will be explained in Section 3.1.1., interactive consistency algorithms are often to be preferred to consensus protocols. In Section 3.1.3., it is explained that the design of an interactive consistency algorithm (or ***ICA***) boils down to the design of a so-called ***Byzantine Agreement Protocol*** (which will be defined in Section 3.1.3.). The focus of this chapter will therefore be on Byzantine Agreement Protocols (or ***BAPs*** for short). Byzantine Agreement Protocols were originally defined in [LaSP82].

Many different Byzantine Agreement Protocols satisfy the definition in Section 3.1.3. In Section 3.1.4., the protocols are classified according to some important system and protocol parameters. In general, these protocols require much communication overhead and a large number of processors. In this chapter we will focus on so-called ***authenticated BAPs*** (defined in Section 3.1.4.), because they require less communication overhead and fewer processors than so-called ***non-authenticated BAPs*** (defined in Section 3.1.4.). Still, however, an authenticated BAP requires a considerable amount of message communication. In Section 3.1.5., we discuss various ways in which the communication overhead required by the original authenticated BAP (described in [LaSP82] and presented in Section 3.1.5.) can be reduced, and we propose a new class of authenticated BAPs[2], in which reduction of the communication overhead is effectuated by reducing the size of messages that are communicated. A more in-depth investigation of these protocols can be found in Section 3.2. In Section 3.1.6., we introduce so-called ***authenticated self-synchronizing BAPs***[3], that work under less strict synchronicity assumptions than existing authenticated BAPs. The details of these authenticated self-synchronizing BAPs will be given in Section 3.3.

Finally, in Section 3.1.7., authenticated self-synchronizing ICAs are introduced, which are based on the authenticated self-synchronizing BAPs presented in Section 3.1.6. The details of these authenticated self-synchronizing ICAs are given in Section 3.4.

3.1.1. Consensus protocols

In literature, several different definitions of a consensus protocol exist. For an overview of consensus protocols, see, e.g., [Alst96, BaDo88, BaMD93, TuSh92]. An extensive bibliography on consensus protocols can be found in [Coan90].

In [DwLS88], a ***consensus protocol*** is defined as follows.

Let N be a set of N processors p_i ($1 \le i \le N$). Every processor p_i possesses an initial value v_i from some arbitrary value domain[4] V. In the presence of up to T faulty processors, the consensus protocol guarantees ***consensus***, i.e., satisfaction of the following three conditions:

C1. All correct processors decide on the same value.

C2. If all initial values v_i are equal, say v, then all correct processors decide v.

2. These protocols have been published in [PoKr95]
3. These BAPs have been published in [PoKr96] and [PoKM97]
4. In the definitions of consensus protocols in [Alst96], set V may be any totally ordered set.

C3. All correct processors eventually decide.

The consensus protocols in [BaDo88, BDGK95, BeGP89, BeGP92, CoWe92, GaMo93] also satisfy the definition above, for a binary value domain V (i.e., $V = \{0,1\}$).

A number of variants to the above definition exist. E.g., in [Alst96], consensus protocols are defined which satisfy a slightly different second condition. A distinction is made between weak consensus protocols and strong consensus protocols respectively.

In [Alst96], a ***weak consensus protocol*** is defined as a protocol that guarantees ***weak consensus***, i.e., satisfaction of conditions C1 and C2', and C3, where C2' is defined by:

C2'. If all processors are correct, all processors decide on the initial value of one of the processors p_i.

Furthermore, in [Alst96], a ***strong consensus protocol*** is defined as a protocol that guarantees ***strong consensus***, i.e., satisfaction of condition C1, C2", and C3, where C2" is defined as follows:

C2". The decision of a correct processor is equal to the initial value of one of the processors p_i.

The definition of the consensus protocol in [Ponz91] is identical to that of the strong consensus protocol given above.

Notice that, in the consensus protocol in [DwLS88], as well as in the weak consensus protocol in [Alst96], it is possible that the decision of the correct processors is different from any processor's initial value (i.e., the decisions may be random values). In a strong consensus protocol, a correct processor always decides a value that is equal to the initial value of one of the processors p_i.

Consensus protocols guarantee that the decisions of correct processors are always equal (by C1). However, it is important to notice that, in **all** consensus protocols mentioned above, *it may be unknown* **which** value is decided by the correct processors. (In the strong consensus protocol and the consensus protocol in [DwLS88], this happens if there exist two or more different initial values. In the weak consensus protocol, it also occurs if one or more processors are faulty.)

For most applications, this will be undesirable. E.g., if the initial value of each processor is defined to be equal to the result of reading (one of) its sensor(s), then it is very well possible that the initial values of the various processors differ from each other, even if all processors function correctly. In this case, one would probably prefer the decision of the correct processors being based on the set of all initial values, instead of being equal to a single arbitrarily chosen initial value (which may even be the initial value of a faulty processor).

The decision taken in a correct processor can be based on the initial values of all processors if the processors execute a so-called ***interactive consistency algorithm***. These algorithms will be defined in the next section. An interactive consistency algorithm does not specify which function is applied to the set of initial values, it only guarantees that all correct processors agree on the set of initial values. However, since in an interactive consistency algorithm, it is possible to base the decision taken in the correct processors on the initial values of all processors, interactive consistency algorithms are often to be preferred to consensus protocols.

3.1.2. Interactive consistency algorithms

Interactive consistency algorithms were originally defined in [PeSL80]. In terms of communicating processors, the algorithm can be defined as follows.

Let N be a set of N processors p_i ($1 \leq i \leq N$). Every processor p_i possesses an initial value v_i. The initial values are distributed from all processors to all processors by means of an interactive consistency algorithm. In the presence of up to T faulty processors, an interactive consistency algorithm guarantees ***interactive consistency*** [PeSL80], i.e., satisfaction of the following two conditions:

ICA1. All correct processors agree on the initial value they think they have received from each of the processors.

ICA2. For every processor p, it holds that if p is correct, the above-mentioned agreement equals the initial value actually sent by processor p.

In [LaSP82], an interactive consistency algorithm consists of execution of a number of so-called ***Byzantine Agreement Protocols***, one for each processor. In [LaSP82], in terms of communicating processors, such a Byzantine Agreement Protocol is defined as follows.

Let N be a set of N processors p_i ($1 \leq i \leq N$). A certain processor p_i possesses an initial value v_i. The initial value v_i is distributed from processor p_i to all other processors by means of a Byzantine Agreement Protocol. In the presence of up to T faulty processors, a Byzantine Agreement Protocol guarantees ***Byzantine Agreement***, i.e., satisfaction of the so-called ***interactive consistency conditions*** [PeSL80, LaSP82]:

IC1. All correct processors agree on the initial value they think they have received from processor p_i.

IC2. If processor p_i is correct, the above-mentioned agreement equals the initial value actually sent by processor p_i.

In [Alst96], a slightly more restrictive definition of Byzantine Agreement is used. In [Alst96, p.18], an algorithm is said to solve Byzantine Agreement, iff it satisfies IC1, IC2, and IC3, where IC3 is defined by

IC3. All correct processors eventually decide.

In this chapter, we will use the original definition of Byzantine Agreement given in [LaSP82], since this definition is more widely used than the definition given in [Alst96].

In the next section, Byzantine Agreement Protocols will be treated in more detail.

3.1.3. Byzantine Agreement Protocols

A Byzantine Agreement Protocol solves the so-called ***Byzantine Generals Problem***. This problem was defined in [LaSP82] and named after a story used to explain the problem of reaching agreement in the presence of faults. The story (cited from [LaSP82]) is as follows.

“We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement. The generals must have an algorithm to guarantee that:

A. All loyal generals decide upon the same plan of action.

The loyal generals will all do what the algorithm says they should, but the traitors may do anything they wish. The algorithm must guarantee condition *A* regardless of what the traitors do. The loyal generals should not only reach agreement, but should agree on a reasonable plan. We therefore also want to insure that:

B. A small number of traitors cannot cause the loyal generals to adopt a bad plan.”

Here, we finish quoting the story from [LaSP82].

Condition *B* is hard to formalize, since it requires defining what a bad plan is, and we do not attempt to do so. Instead, we focus on how the generals reach a decision. The above-sketched problem is therefore split into the following two parts:

❑ Initially, all generals possess a part of the data, the initial data, on which the plan will be based. The initial data is distributed by all generals to all generals by means of an ***interactive consistency algorithm*** (originally defined in [PeSL80]). An interactive consistency algorithm solves the so-called ***interactive consistency problem*** [PeSL80], i.e., it satisfies the following two conditions:
  - all loyal generals agree on the data they think they have received from each of the generals.
  - for any general *g*, it holds that if general *g* is loyal, the above-mentioned agreement equals the initial data actually sent by general *g*.

  The interactive consistency problem can be solved by having every general *g* distribute his/her initial data to all other generals by means of a ***Byzantine Agreement Protocol***, which ensures that:
  - all loyal generals agree on the data they think they have received from general *g*.
  - if general *g* is loyal, the above-mentioned agreement equals the initial data actually sent by general *g*.

  Thus, an interactive consistency algorithm may be designed as an algorithm in which every general executes a BAP on his/her initial data [LaSP82].

❑ After having executed an interactive consistency algorithm, all loyal generals possess the same data (the data of a general *g* is called the ***interactive consistency***

vector [PeSL80] of general *g*), on which they apply the same algorithm (whatever this may be) in order to come to a good plan.

So, the above-sketched problem boils down to designing a BAP. We now abstract from the story of the Byzantine generals and consider the problem of reaching agreement among correct processors in a system of communicating processors, a number of which may be faulty.

A BAP runs on a system consisting of *N* processors. One of the processors in the system is the ***source***, the others are ***destination processors***. The role of the source is played by a replica that wants to communicate an input value which it received from a client. The other processors in the system service play the role of a destination processor. An extensive overview of BAPs is given in [BaMD93].

A BAP consists of a multicast process and a decision-making process.

In the ***multicast process***, in a number of communication phases, a message value is transmitted from the source to a set of destination processors. In every communication phase, processors relay the messages they have received in the previous communication phase via communication links to other processors. So, in every communication phase of the multicast process, one or more processors send one or more messages to one or more other processors. The event of one processor sending one message to one other processor may occur multiple times in each communication phase. In such an event, the processor that sends the message is referred to as the ***sender***, and the processor that receives the message as the ***receiver***. The terms sender and receiver should not be confused with the terms source and destination processor. The sender of a message may either be the source or a destination processor, and each destination processor may both be sender and receiver of a message in each communication phase.

In the ***decision-making process***, every destination processor calculates a decision about what the source has sent on basis of the messages it has received during the multicast process. Up to *T* processors (i.e., the source and / or destination processors) in the system may behave maliciously. Regardless which processors are faulty and which data was sent by the source, the protocol guarantees ***Byzantine Agreement***, i.e., satisfaction of the ***interactive consistency conditions*** [PeSL80, LaSP82]:

IC1. *The correct processors agree on the data they think they have received from the source.*

IC2. *If the source is correct, the above-mentioned agreement equals the data sent by the source.*

What makes the problem of guaranteeing Byzantine Agreement in the presence of arbitrarily faulty processors so difficult, is the faulty processors' ability to behave maliciously. A faulty processor need not behave in the manner specified: it may refuse to send or relay messages, it may send or relay messages at arbitrary times (different from those specified in the BAP) and furthermore, it may corrupt the contents of a message before relaying it

.

BAPs

- **randomized BAPs** (*asynchronous systems, ρ or τ_{max} non-existent*)

  [Rabi83, BenO83, BrTo83], and many others

- **deterministic BAPs** (*synchronous systems, ρ and τ_{max} exist, ρ and τ_{max} a priori known, ρ and τ_{max} hold during BAP*)

  - **non-authenticated BAPs**

	processor clocks must be exactly synchronized	processor clocks must be approximately synchronized	processor clocks need not be synchronized
start of protocol must be synchronized	[PeSL80, LaSP82, Krol91, Krol95] and many others		
protocol is self-synchronizing			

  - **authenticated BAPs**

	processor clocks must be exactly synchronized	processor clocks must be approximately synchronized	processor clocks need not be synchronized
start of protocol must be synchronized	[PeSL80, LaSP82, GoLR95, DoSt83, PoKr95], and many others		
protocol is self-synchronizing		[CASD95]	[PoKr96, PoKM97]

- **partially synchronous BAPs** (*synchronous systems, ρ and τ_{max} exist, ρ or τ_{max} not a priori known, or ρ or τ_{max} only hold starting at some unknown time t)*

  No examples found in literature

Figure 3.1. Classification of BAPs

3.1.4. Classification of Byzantine Agreement Protocols

The design of a BAP depends on the assumptions made for the BAP and for the system in which the BAP is executed. In Figure 3.1., existing BAPs are classified according to the following protocol and system model parameters:

- ❑ the synchrony of the system
- ❑ message authentication
- ❑ clock skew between clocks of correct processors
- ❑ amount of protocol synchronization at the start of the BAP

Besides these parameters, BAPs are characterized by several other important parameters, such as the number of processors, *N*, the maximum number of faulty processors, *T*, for up to which the BAP guarantees Byzantine Agreement, and the number of communication phases, *K*. Parameter *T* is a design parameter, which may be freely chosen. The minimum values for *N* and *K* are determined by the selected value of *T*, and of the above-mentioned protocol and system model parameters.

In this section, BAPs will be classified with regard to the synchrony of the system and with regard to message authentication. In Section 3.1.6., for authenticated BAPs, a classification is made with regard to the clock skew between clocks of correct processors and with regard to the amount of protocol synchronization at the start of the BAP.

From Figure 3.1., it is apparent that several types of BAPs have not yet appeared in literature (e.g., partially synchronous BAPs, or deterministic *non-authenticated* BAPs for which the processor clocks need not exactly be synchronized, or for which the start of the BAP need not exactly be synchronized). An interesting question is whether these BAPs do actually exist or not. An answer to this question is beyond the scope of this thesis, however.

3.1.4.1. Classification of BAPs according to the synchrony of the system

An important system model parameter is the synchrony of the system. In literature, synchronous and asynchronous systems are distinguished. In [DoDS87], several synchrony parameters are identified, resulting in different definitions of a synchronous system.

In this chapter, we define a ***synchronous system*** as a system with the following two properties:

1. The rate at which the processor clock of any correct processor drifts from real-time, is bounded by a factor (1+ρ). Such clocks are called ρ-***bounded clocks*** [LuLy88]. Processor clocks of faulty processors may run at arbitrary rates.
2. Furthermore, there exists a real-time upper bound (τ_{max}) on the time needed to communicate a message from a correct processor to another processor in the system.

Conversely, any system that is not synchronous is an ***asynchronous system***.

The definitions of synchronous and asynchronous systems given above are very similar to those commonly used in research on Byzantine Agreement[5].

Different types of BAPs exist for both types of systems. For asynchronous systems, so-called ***randomized BAPs*** are designed. These randomized BAPs were first studied in

[Rabi83, BenO83, BrTo83], for an overview, see [BaMD93]. ***Deterministic BAPs*** (first defined in [PeSL80, LaSP82], see [BaMD93] for an overview) are designed for synchronous systems.

In deterministic BAPs found in literature, it is generally assumed that the bounds $(1+\rho)$ and τ_{max} are known a priori and hold during execution of the BAP. In [DwLS88], so-called partially synchronous systems are introduced. In [DwLS88], a ***partially synchronous system*** is defined as a synchronous system in which the bounds $(1+\rho)$ and τ_{max} exist, but at least one of these bounds is unknown, or at least one of these bounds is only guaranteed to hold starting at some unknown time t. In [DwLS88], several *consensus protocols* are presented for partially synchronous systems. An interesting question is whether it is also possible to design *BAPs* for partially synchronous systems (In Figure 3.1., these BAPs are referred to as ***partially synchronous BAPs***). Answering this question is beyond the scope of this thesis, however. Hence, partially synchronous BAPs will not be further investigated.

Whereas *deterministic* BAPs guarantee Byzantine Agreement within a finite number of communication phases, *randomized* BAPs may need an infinite number of communication phases to complete. In each communication phase of a randomized BAP, there is a probability that faulty processors thwart Byzantine Agreement [BaMD93]. Thus, randomized BAPs cannot guarantee Byzantine Agreement within finite time. It is even possible to prove that deterministic Byzantine Agreement within finite time is impossible in a ***completely asynchronous system*** (i.e., a system in which no bounds $(1+\rho)$ and τ_{max} exist, and in which the rate at which a processor clock of a correct processor drifts from real-time, or the time needed to communicate a message from a correct processor to another processor in the system may actually be infinite), as follows.

In [FiLP85], Fischer et al. proved that (deterministic) *consensus* is impossible in asynchronous systems. The impossibility result in [FiLP85] applies to a very weak form of the consensus problem[6]. It implies that, in a completely asynchronous system, there exists no protocol that guarantees satisfaction of conditions C1, C2 (or C2' respectively C2"), and C3 from Section 3.1.1. The latter implies that deterministic *Byzantine Agreement* within finite time is also impossible in a completely asynchronous system. This is proved as follows.

Assume that in a completely asynchronous system, there exists a protocol *P* that guar-

5. In [DoDS87], Dolev et al. state that the following definitions for synchronous and asynchronous systems are commonly used in research on Byzantine Agreement:

   A ***synchronous*** system is defined as a system in which:
   1. Correct processors are synchronous, i.e., there is a fixed known upper bound on the relative speeds of correct processors.
   2. Communication is synchronous, i.e., there is a fixed known upper bound on the time required for a message to be sent from one processor to another.

   Any system that is not synchronous is an ***asynchronous*** system.

   It can easily be seen that our definitions of synchronous respectively asynchronous systems only differ slightly from the definitions given above. Our definition of a synchronous system is slightly more restrictive than the definition above. In addition to the definition above, we require the correct processors to have a bounded drift from ***real-time***.

antees satisfaction of the interactive consistency conditions within finite time. We prove that, in this case, there would also exist a strong consensus protocol, i.e., a protocol that guarantees satisfaction of C1, C2”, and C3. Let *F* be some deterministic function, which is defined on a set of values, *S*, such, that application of function *F* on *S* yields one of the elements of *S* as a result. Notice that there exist many functions *F* that satisfy this specification (e.g., selection of the minimum value or the maximum value in *S*). We define a protocol *Q*, in which all correct processors execute protocol *P*, and, after having performed protocol *P*, every correct processor applies function *F* on the elements in its interactive consistency vector. Then, after execution of protocol *Q*, condition C1, C2”, and C3 would be satisfied. However, we know that there exists no protocol that guarantees satisfaction of condition C1, C2”, and C3. Thus, protocol *Q* cannot exist, and hence, protocol *P* cannot exist.

Thus, we must conclude that, in a completely asynchronous system, there exists no protocol that guarantees satisfaction of the interactive consistency conditions within finite time. Hence, deterministic Byzantine Agreement within finite time is impossible in a completely asynchronous system.

Furthermore, Dolev et al. [DoDS87], building on the work of Fischer et al. [FiLP85], proved that if either a fixed upper bound on message delivery does not exist (i.e., communication is asynchronous) or a fixed upper bound on relative processor speeds does not exist (i.e., processors are asynchronous), then there is no consensus protocol resilient to even one fail-stop fault [DwLS88].

Since, in a dependable distributed system, it is important to be able to guarantee Byzantine Agreement within finite time, our focus will be on *deterministic* BAPs in a *synchronous* system. We assume a synchronous system in which the bounds τ_{max} and $(1+\rho)$ are a priori known by each correct processor in the system. Furthermore, we assume that there also exists a real-time lower bound (τ_{min}) on the time needed to communicate a message from a correct processor to another processor in the system. We assume that τ_{min} is a priori known by each correct processor.

3.1.4.2. Classification of deterministic BAPs according to message authentication

In literature, for *deterministic* BAPs, a distinction is made between authenticated and non-authenticated BAPs. In a so-called ***authenticated BAP***, the faulty processors’ ability to behave maliciously is restricted by having every correct processor sign each message it relays with an unforgeable ***signature***. In a ***non-authenticated BAP***, no signatures are applied.

---

6. In the consensus protocol proposed by Fischer et al. in [FiLP85], it is assumed that every process starts with an initial value in {0,1}. Then, the proposed consensus protocol should guarantee satisfaction of the following conditions:
   1. All correct processors decide on the same value.
   2. If a correct processor decides, it decides on a value in {0,1}.
   3. Some correct processor eventually decides.

   Notice that this set of conditions is weaker than that of any of the consensus protocols presented in Section 3.1.1., i.e., satisfaction of the conditions of one of the consensus protocols presented in Section 3.1.1. implies satisfaction of the conditions above.

Deterministic *non-authenticated* BAPs only exist for $N \geq 3T+1$ and $K \geq T+1$. These bounds are proved in [PeSL80, LaSP82] and [FiLy82] respectively. Deterministic *authenticated* BAPs can tolerate an arbitrary number of faulty processors. These protocols need only satisfy $K \geq T+1$ (proved in [FiLy82]).

In this chapter, our focus will be on authenticated BAPs, because of their lower communication overhead and lower number of processors required, if compared to non-authenticated BAPs.

In authenticated BAPs, it is usually assumed that a correct processor's signature cannot be forged and that any alteration of the contents of its signed messages can be detected. No assumptions are made about a faulty processor's signature. In particular, ***collusion*** ([LaSP82]) among faulty processors is permitted, i.e., a faulty processor's signature may be forged by other faulty processors.

In this chapter, we assume that every correct processor possesses an unforgeable signature. The problem of distribution of these signatures among the processors is not trivial, however, solving this problem is beyond the scope of this chapter.

It is impossible to implement unforgeable signatures. However, they can be approximated by means of cryptographic techniques, e.g., error-detecting codes [VaOo89] or public-key cryptosystems [DiHe76, RiSA78] based on digital signatures. Public-key cryptosystems may yield a better approximation of unforgeable signatures than error-detecting codes, be it, however, at the cost of a larger computational overhead. Gong et al. (in [GoLR95]) mention several difficulties that are encountered when one tries to implement unforgeable signatures. Nevertheless, in this chapter we will assume that the signatures of correct processors are unforgeable [7]. This is not a very restrictive assumption, since, by increasing the length of the cryptographic key or the length of the error-detecting code, and taking into account the precautions suggested in [GoLR95], the probability of a correct processor's signature being forged can be made arbitrarily small [LaSP82, p.400].

Any correct processor should change its unforgeable signature from time to time for the following reason. From each message that has been signed with the unforgeable signature of some correct processor *c*, some information about *c*'s unforgeable signature can be obtained. A malicious processor may try to forge the unforgeable signature of processor *c* by analyzing as many messages signed by processor *c* as it is able to obtain. The more of these signed messages a malicious processor *f* has obtained, the greater the probability that processor *f* is able to forge *c*'s signature. By changing its unforgeable signature from time to time, processor *c* can prevent any malicious processor from obtaining too many messages signed with the same unforgeable signature, and hence, from forging its unforgeable signature.

The first deterministic authenticated BAP was described by Lamport in [LaSP82]. This BAP will be presented in the next section. Although the communication overhead for an *authenticated* BAP is much smaller than in a *non-authenticated* BAP, the communication overhead in the original authenticated BAP described by Lamport is still quite

7. Violation of this assumption (i.e., forgery of a correct processor's signature) may cause violation of the interactive consistency conditions.

large. Furthermore, the original BAP described by Lamport has tight requirements with regard to the synchrony of the system.

In the remainder of this chapter we focus on methods to reduce the communication overhead in an authenticated BAP, and we describe ways to adapt existing authenticated BAPs in order to guarantee Byzantine Agreement in a system with less strict synchronicity assumptions.

3.1.5. Reduction of the communication overhead of an authenticated BAP

The first authenticated BAP appeared in [LaSP82]. The protocol, called SM (from ‘Signed Messages’), was originally defined in terms of a commander who gives a (signed) order to a number of lieutenants. By exchanging messages, the lieutenants must reach agreement about the order that the commander has sent. Below, we will describe the SM-protocol in terms of communicating processors. Clearly, in these terms, the commander corresponds to the source, and the lieutenants to the destination processors in the system.

The SM-protocol is executed by a set N of processors, one of which is the source. The multicast process of the SM-protocol consists of T+1 communication phases. In the notation used in the SM-protocol, a distinction is made between a message and its so-called ***message value*** (i.e., the information that the sender of the message wants to communicate). With $x{:}i$, we denote a message which contains the message value x signed by processor i ($i \in N$). Thus, $v{:}j{:}i$ denotes the message value v signed by j, and then that value $v{:}j$ signed by i ($i,j \in N$ and $i \neq j$). Processor 0 is the source ($0 \in N$). Each processor $i \in N$ maintains a set V_i, containing the set of message values from properly signed messages obtained thus far. In [LaSP82], the protocol is defined as follows:

Protocol SM(T), T > 0
Initially, for each processor $i \in N$, $V_i = \emptyset$.

```
(1)  The source signs and sends the message $v{:}0$ to every processor $i \in N \setminus \{0\}$. Furthermore, it adds $v$ to $V_0$.
(2)  For each processor $i \in N$:
     (a)  If processor $i$ receives a message of the form $v{:}0$ from the source and $V_i = \emptyset$, then:
          (1) it lets $V_i$ equal $\{v\}$
          (2) it signs and sends the message $v{:}0{:}i$ to every processor $j \in N \setminus \{0,i\}$.
     (b)  If processor $i$ receives a message of the form $v{:}0{:}j_1 \ldots j_k$, and $v$ is not in the set $V_i$, then
          (1) it adds $v$ to $V_i$
          (2) if $k < T$, then it signs and sends the message $v{:}0{:}j_1{:}\ldots{:}j_k{:}i$ to every processor $j \in N \setminus \{0, j_1 \ldots j_k\}$
(3)  For each processor $i \in N$:
     When processor $i$ will receive no more messages, it decides choice($V_i$) (where choice is a function on the message values in $V_i$. If $V_i$ consists of a single message value $v$, then choice($V_i$) will be equal to $v$.)
```

It is easy to prove that, for any T, the protocol $SM(T)$ solves Byzantine Agreement if the system contains at most T faulty processors, provided that the following assumptions (stated in [LaSP82]) hold:

La1. Every message that is sent is delivered correctly.
La2. The receiver of a message knows who sent it.
La3. The absence of a message can be detected.
La4. (a) A correct processor's signature cannot be forged, and any alteration of the contents of its signed messages can be detected.
(b) Any processor can verify the authenticity of the signature of the source.

The proof given in [LaSP82, p.393] is as follows:

Proof:
We first prove IC2. If the source 0 is correct, then it sends its signed message v:0 to every processor $i \in N \setminus \{0\}$ in step (1). Every correct processor i will therefore receive the message value v in step (2)(a). Moreover, since no faulty processor $j \in N \setminus \{0\}$ can forge any other message of the form v':0, a correct processor $i \in N \setminus \{0\}$ can receive no additional message value in step (2)(b). Hence, for each correct processor $i \in N \setminus \{0\}$, the set V_i obtained in step (2) consists of the single message value v, which will be decided in step (3). This proves IC2.

Since IC1 follows from IC2 if the source is correct, to prove IC1, we need only consider the case in which the source is faulty. Two correct processors i and j, with $i, j \in N \setminus \{0\}$, decide the same value in step (3) if the sets of message values V_i and V_j that they receive in step (2) are the same. Therefore, to prove IC1, it suffices to prove that, if i puts a message value v into V_i in step (2), then j must put the same message value v into V_j in step (2). To do this, we must show that j receives a properly signed message containing that message value. If i receives the message value v in step (2)(a), then i sends it to j in step (2)(a)(2), so j receives it (by La1). If i adds message value v to V_i in step (2)(b), then i must receive a first message of the form $v{:}0{:}j_1{:}\ldots{:}j_k$. If j is one of the j_r ($1 \leq r \leq k$), then, by La4, j must already have received the message value v. If not, we consider two cases:

1. $k < T$. In this case, i sends the message $v{:}0{:}j_1{:}\ldots{:}j_k{:}i$ to j, so j must receive the message value v.
2. $k = T$. Since the source 0 is faulty, at most T-1 of the processors in $N \setminus \{0\}$ are faulty. Hence, at least one of the processors $j_1, \ldots, j_T$ is correct. This correct processor must have sent j the value v when it first received it, so j must therefore receive that value.

This completes the proof. ❑

In protocol SM(T), in each communication phase of the multicast process of the SM-protocol, all correct processors relay the messages they receive to *all* processors that did not yet receive it. As a result, the SM-protocol is rather inefficient with regard to the required number of messages. Since then, a vast amount of literature has appeared on BAPs that are more efficient with regard to the required communication overhead.

Different strategies can be applied in order to reduce the required communication overhead:

- ❏ reduction of the number of messages relayed by a correct processor.
- ❏ reduction of the number of destination processors which a message is relayed to.
- ❏ reduction of the size of the messages.

For authenticated BAPs, the first two strategies to reduce the communication overhead were originally proposed in [DoSt83].

The first reduction strategy is established in [DoSt83] by ensuring that no correct processor relays more than two messages. A correct processor only relays a message if it has not relayed a message with the same message value before. Furthermore, if it has relayed two different messages with distinct message values, it does not relay any subsequent messages. In the latter case, at the end of the protocol, it may conclude that the source was faulty.

The second reduction strategy, as proposed in [DoSt83], consists of dividing the processors in the system into so-called ***relay processors*** and ***non-relay processors***. Then, at random, T+1 processors in the system are chosen to be relay processors, the others are non-relay processors. Now, any non-relay processor is required to send messages only to relay processors. Relay processors send messages to all processors that did not receive the message before. The source is considered to be a non-relay processor. In [DoSt83], this reduction method is combined with the first one, by again ensuring that no correct processor relays more than two messages.

More details about these two reduction strategies can be found in [DoSt83].

In Section 3.2., we describe a new class of authenticated BAPs in which the third reduction strategy is applied, i.e., the size of the messages that are communicated during the multicast process is reduced. This is established by having correct processors *encode* messages into symbols of an ***erasure-correcting code***[8] instead of having them *multicast* these messages as in the original authenticated BAP described by Lamport et al. in [LaSP82]. In [Krol91, Krol95], this reduction strategy was invented and elaborated for non-authenticated BAPs, however, in this chapter, the strategy is proven to be equally well applicable for authenticated BAPs. The authenticated BAPs described in Section 3.2. have appeared previously in [PoKr95].

3.1.6. Guaranteeing Byzantine Agreement under less strict synchronicity assumptions

Many deterministic authenticated BAPs are based on the assumption of guaranteed communication in a network of perfectly synchronized processors (e.g., in [PeSL80,

8. An erasure-correcting code is a special kind of error-correcting code. Provided that data is encoded into symbols of an erasure-correcting code, it is possible to automatically correct a number of ***erasures***, i.e., a number of mutilated symbols of which the position in the code is known. Since all valid messages (i.e., symbols of the erasure-correcting code) that are communicated are signed with unforgeable signatures, any mutilation of a symbol can be detected, and thus, it is known which symbols are mutilated.

LaSP82, GoLR95, DoSt83, PoKr95]). They assume that communication takes place in synchronous rounds of information exchange. These protocols assume a synchronous system in which $\rho = 0$ and $\tau_{min} = \tau_{max}$ (i.e., all processors run at the same rate, and message delivery times are fixed). Such a synchronous system is called a ***lock-step synchronous system***. Furthermore, in these protocols, it is assumed that all processors know the start of the protocol and start the protocol simultaneously.

In reality, processors are not automatically synchronized, nor do the correct processors a priori know the start of a BAP. It seems easy to solve these problems by having all correct processors reach ***exact agreement*** about some point in time.

In a distributed system, however, reaching exact agreement about a common point in time is a difficult problem. Since every processor has its own local clock, processors do not automatically have a common notion of time. The correct processors may obtain a common notion of time by executing a so-called ***fault-tolerant clock synchronization algorithm*** (see [SiLL90] for an overview). However, due to uncertain message delivery times, and processor clocks running at differing rates, any fault-tolerant clock synchronization algorithm can only establish *appromixate* synchronization between processor clocks[9]. This makes it hard to have all correct processors reach exact agreement about a common point in time.

Exact agreement about some point in time can be obtained by means of a ***Distributed Firing Squad (DFS) algorithm***[10] [CDDS85]. However, a DFS-algorithm cannot be used to solve the problem, since, in order to function correctly in the presence of arbitrarily faulty processors, a DFS-algorithm requires that the clocks of all correct processors run at the same rate, and that messages communicated between certain processors take a fixed time to be delivered. In a distributed system, it is hard to establish the amount of synchronism required by the DFS-algorithm, because, in practice, no two processor clocks run at the same rate [LaSP82], and message delivery times are uncertain.

In [SuHW94], it is proposed to run a BAP in order to have all correct processors obtain (as a result) exact agreement about a common point in time. Clearly, this method is not applicable here, because correct processors must already have obtained exact agreement about a common point in time *before* a BAP can be executed.

9. In [LuLy84], Lundelius and Lynch prove that in a completely connected network of N processors, a lower bound on the tightness of synchronization (i.e., the minimal clock skew) between the processor clocks of the correct processors in the system is $\eta \cdot (1 - 1 / N)$, where η is the difference between the bounds on the message delay (i.e., $\eta = \tau_{max} - \tau_{min}$ in our definition of a synchronous system in Section 3.1.4.1.). The work in [LuLy84] was extended to arbitrary networks in [HaMM85].
10. Exact agreement about a point in time can be obtained by executing a DFS-algorithm on a system of N processors, since in the presence of up to T maliciously behaving processors, such an algorithm satisfies the following two properties:

DFS1.	If any correct processor receives a message to start a DFS synchronization, then at some future time all correct processors will “fire” (formally, enter a special state), and
DFS2.	The correct processors all fire simultaneously.

More details about the DFS-algorithm can be found in [CDDS85].

Since, in a distributed system, it is so difficult to have all correct processors reach exact agreement about a common point in time, some authenticated BAPs (see, e.g., [CASD95]) have been designed which guarantee Byzantine Agreement provided that, at the start and during execution of the BAP, the clocks of all correct processors are ***approximately synchronized***. In other words, at the start and during execution of the BAP, the ***clock skew*** (i.e., the difference in clock values [SuHW94]) between the clocks of all correct processors must be within certain bounds. In these BAPs, the processors need not know the start of the protocol.

As stated before, correct processors may become approximately synchronized by running a ***fault-tolerant clock synchronization algorithm***. A great number of fault-tolerant clock synchronization algorithms have appeared in literature (see [SiLL90] for an overview). Roughly, these algorithms can be divided into two groups: probabilistic and deterministic algorithms, respectively. *Probabilistic* fault-tolerant clock synchronization algorithms (e.g., [Cris89]) reach better average synchronization between processor clocks than deterministic algorithms [Jalo94, p.97]. However, in probabilistic algorithms there is always a probability that the clocks cannot be synchronized within finite time [Jalo94]. Thus, running a probabilistic fault-tolerant clock synchronization algorithm cannot guarantee that the synchronization that is required in order to be able to execute a Byzantine Agreement Protocol can be obtained in finite time. Since in a dependable distributed system, any Byzantine Agreement Protocol should terminate in finite time, probabilistic fault-tolerant clock synchronization algorithms are not applicable here. Deterministic fault-tolerant clock synchronization algorithms can guarantee approximate synchronization within finite time. These algorithms require that $N \geq 3T+1$, unless authentication is used or there is a bound on the rate at which messages can be generated (proved in [DoHS86]). In general, it is impossible to indicate a priori an upper bound on the rate at which messages can be generated, so either we must require that $N \geq 3T+1$, or authentication must be used.

Clock synchronization algorithms that use authentication (e.g., the algorithm in [DHSS95]) can tolerate an arbitrary number of failures, provided that the clocks of all correct processors are initially approximately synchronized. In order to keep the clocks of correct processors approximately synchronized, the clocks must be periodically resynchronized, even if all processors function correctly. This is due to uncertain message delivery times, and processor clocks running at different rates (See [SuHW94] for details). In fact, many fault-tolerant clock synchronization algorithms only perform periodical resynchronization of initially approximately synchronized clocks [DHSS95]. These algorithms may be able to tolerate any number of faulty processors (see, e.g., the algorithm proposed in [DHSS95]). However, these algorithms are not capable to integrate new or repaired processors such that their clocks become approximately synchronized with those of the other correct processors (that were initially approximately synchronized).

To integrate (join) new or repaired processors such that their clocks become synchronized with those of all the other correct processors, a so-called ***bounded joining algorithm*** [DHSS95] is required. A necessary condition for a bounded joining algorithm to be guaranteed to succeed is that a majority of the processors in the system be correct (i.e., $N \geq 2T+1$) (proved in [DHSS95]).

So, for the clocks of all correct processors to remain approximately synchronized in the presence of an arbitrary set of up to T faulty processors, which may differ for each run of the BAP in [CASD95], a deterministic fault-tolerant clock synchronization algorithm must be periodically executed (in order to perform periodic resynchronization of the processor clocks), and, moreover, the system must contain at least a majority of correct processors (since execution of a bounded joining algorithm may be required).

In fact, for an authenticated BAP to work, it is sufficient that during execution of the BAP, approximate ***protocol synchronization*** exists between the correct processors in the system. I.e., it is sufficient that the protocol is approximately simultaneously executed on all correct processors in the system. Approximate clock synchronization between the processor clocks of correct processors, as is required at the start and during execution of the BAPs in [CASD95], is only a means to establish approximate protocol synchronization between the correct processors during execution of the BAP. The price that has to be paid to establish approximately synchronized clocks is high: the system should contain at least a majority of correct processors, which should periodically execute a fault-tolerant clock synchronization algorithm.

The new class of ***authenticated self-synchronizing BAPs*** which will be presented in Section 3.3. show that it is also possible to establish approximate protocol synchronization between the correct processors in the system *during execution of the BAP*, without requiring the clocks of correct processors to be approximately synchronized at the start or during execution of the BAP. These BAPs guarantee Byzantine Agreement in a synchronous system, regardless of the number of faulty processors in the system, while allowing arbitrary clock skew between the processor clocks of the processors in the system. In these BAPs, the processors need not know the start of the protocol. The required amount of protocol synchronization between the correct processors in the system is established during execution of the BAP itself. In Section 3.3.1., we will describe how protocol synchronization between the correct processors in the system is achieved in our BAPs. The processor clocks are *not* synchronized by execution of these BAPs.

However, the above-mentioned authenticated self-synchronizing BAPs are inefficient with regard to the number of messages communicated during the BAP. In this chapter, we will therefore also define a new class of ***optimized authenticated self-synchronizing BAPs***, which are based on the previously described authenticated self-synchronizing BAPs, but require less communication overhead.

Another assumption that is commonly made for deterministic authenticated BAPs, is that every correct processor immediately notices when its processor clock reaches a certain predefined value (in particular, a clock value at which a communication phase ends). However, every processor clock is a discrete clock with finite precision, and moreover, a processor may not continuously check the value of its processor clock. Therefore, it is more realistic to assume that a processor can only determine within certain bounds that its processor clock is at or beyond a certain clock value. The authenticated self-synchronizing BAPs presented in this chapter will be based on this so-called ***imprecise clock value measurement assumption***[11].

In Section 3.3.9. and 3.3.10. respectively, we prove that our authenticated self-synchronizing BAPs guarantee Byzantine Agreement in a synchronous system under the imprecise clock value measurement assumption.

A vast amount of literature has appeared on authenticated BAPs, based on varying synchronicity assumptions. In Table 3.1, we have classified several authenticated BAPs according to assumptions made with regard to the required tightness of synchronization (i.e., the maximally allowed clock skew) between the processor clocks of correct processors in the system, and the tightness of synchronization required at the start of the protocol (protocols that require a synchronous start in all correct processors vs. self-synchronizing protocols) Notice that, of all the BAPs referred to in Table 3.1, the authenticated BAPs in [PoKr96, PoKM97], which are self-synchronizing and which allow arbitrary clock skew between the clocks of the processors in the system are *the least restrictive* protocols with regard to the synchronicity that is required. These BAPs are not only presented in [PoKr96,PoKM97], but will also be the subject of Section 3.3.

Table 3.1: Classification of authenticated BAPs with regard to synchronicity assumptions

	processor clocks must be exactly synchronized (*no clock skew allowed*)	**processor clocks must be approximately synchronized** (*bounded clock skew allowed*)	**processor clocks need not be synchronized** (*arbitrary clock skew allowed*)
start of protocol must be synchronized	[PeSL80, LaSP82, GoLR95, DoSt83, PoKr95], BAPs in Section 3.2., and many others		
protocol is self-synchronizing		[CASD95]	[PoKr96, PoKM97], BAPs in Section 3.3.

It is worthwhile noticing that, for ***consensus protocols*** (See Section 3.1.1.), a similar classification can be made as it is done in Table 3.1 for authenticated BAPs. Many consensus protocols (e.g., the protocols in [BaDo88, BDGK95, BeGP89, BeGP92, CoWe92, GaMo93, Alst96]) are designed for a lock-step synchronous system. These protocols assume guaranteed communication in a network of perfectly synchronized processors. Furthermore, they require the correct processors to know the start of the protocol and to start the protocol simultaneously. Since it is hard to satisfy these requirements in a distributed system, a number of protocols have been proposed with less strict assumptions with regard to the synchronicity of the system.

In [Ponz91], consensus protocols are designed for a synchronous system with arbitrary ρ and τ_{max}. However, the consensus protocols in [Ponz91] only work, if all correct processors know the start of the protocol and start the protocol simultaneously.

In [DwLS88], several consensus protocols have been proposed for a ***partially synchronous system***. Partial synchrony lies between the cases of a synchronous system and an

11. Assumption A3.3 in Section 3.3.8

asynchronous system. In a *synchronous system*, there is a known fixed upper bound τ_{max} on the time required for a message to be sent from one processor to another (i.e., communication is synchronous), and a known fixed upper bound $(1+\rho)$ on the relative speeds of different processors (i.e., processors are synchronous). In an *asynchronous system* no fixed upper bounds τ_{max} and $(1+\rho)$ exist. In [DwLS88], consensus protocols are designed for two versions of partial synchrony. In one version of partial synchrony, fixed upper bounds τ_{max} and $(1+\rho)$ exist, but they are not known a priori. In the other version of partial synchrony, the upper bounds are known, but are only guaranteed to hold starting at some unknown time t. The protocols given in [DwLS88] work for arbitrary ρ and τ_{max}.

The consensus protocols in [DwLS88] use so-called fault-tolerant ***distributed clocks*** (which are also described in [DwLS88]). In [DwLS88], distributed clocks are defined as protocols that allow the correct processors to reach some approximately common notion of time. These distributed clocks are used to simulate a lock-step synchronous system in which communication between the processors takes place in synchronized communication rounds. In the proposed consensus protocols, all correct processors start in the same communication round, and any correct processor only proceeds to the next communication round after all messages that should be sent and received respectively by this processor, have indeed been sent and received. In order for these consensus protocols to work, both processors and communication may be partially synchronous (i.e., both bounds $(1+\rho)$ and τ_{max} may be unknown, or only guaranteed to hold after some unknown time t). If communication is partially synchronous, then it is required that $N \geq 3T+1$, if communication is synchronous, then $N \geq 2T+1$ suffices.

3.1.7. Guaranteeing interactive consistency under less strict synchronicity assumptions

As stated in Section 3.1.2., an interactive consistency algorithm (ICA) consists of execution of N BAPs in a system consisting of N processors. In every BAP, a different processor acts as the source in order to communicate its initial value to the other processors.

Most ICAs found in literature assume a lock-step synchronous system in which all N processors know the start of the ICA, and simultaneously start their BAPs belonging to the ICA. The design of an ICA in a lock-step synchronous system is trivial and boils down to simultaneous execution of N BAPs, one per processor.

Typically, a distributed system can be modelled as a synchronous system with arbitrary ρ, τ_{min} and τ_{max} , in which the processors are not automatically synchronized, and do not know the start of the ICA. Although a simultaneous start of the ICA might be obtained by having the correct processors reach exact agreement about a common point in time, this solution is inadequate for a distributed system, since, as stated in Section 3.1.6., in such a system, it is hard to establish lock-step synchronization between the correct processors in the distributed system, and hence, to have the correct processors reach exact agreement about a common point in time.

So, in distributed systems, there exists a need for ICAs that work without requiring the correct processors to reach exact agreement about a common point in time. The

authenticated self-synchronizing ICAs and ***optimized authenticated self-synchronizing ICAs*** introduced in Section 3.4.1. and 3.4.2. respectively, can be used for this purpose, since they do not require the correct processors to reach exact agreement about a common point in time. These ICAs consist of execution of N authenticated self-synchronizing BAPs (introduced in Section 3.3.6.) respectively N optimized authenticated self-synchronizing BAPs (introduced in Section 3.3.7.), and work in a synchronous system, with arbitrary ρ, τ_{min} and τ_{max} , in which the processors are not synchronized and the start of the ICA is not a priori known.

In both types of self-synchronizing ICAs, the required amount of algorithm synchronization between the correct processors in the system is established during execution of the ICA itself.

An authenticated self-synchronizing ICA has been described before in [Nieu91]. The protocol in [Nieu91], referred to as Byzantine Merge Algorithm, is based on a combination of a number of authenticated BAPs (for which protocol synchronization has already been achieved) and a timer activation process (which establishes protocol synchronization). Every processor in the system executes a BAP, in which the duration of the communication phases is measured by a local timer. The Byzantine Merge Algorithm consists of two successively executed parts: the first part establishes protocol synchronization, i.e., approximately simultaneous activation of the timers, whereas the second part consists of approximately simultaneous execution of the BAPs on the different processors in the system. Since the Byzantine Merge Algorithm requires protocol synchronization to be established before the BAPs are started, it is difficult to reduce the communication overhead of this algorithm by means of reducing the number of receivers in each communication phase, as it is done in the authenticated optimized self-synchronizing ICAs.

3.2. Authenticated Byzantine Agreement Protocols based on encoding messages into symbols of an erasure-correcting code

In this section, we will describe a class of deterministic authenticated BAPs which guarantee Byzantine Agreement in a fully-connected lock-step synchronous system for variable-length messages. The protocols have similarities with the SM-protocol described in [LaSP82], which we will refer to as the Lamport-protocol. However, the required amount of data communication is reduced considerably by reducing the number of directions in which data is sent, and, by replacing the multicast functions in the Lamport-protocol by encoder functions of erasure-correcting codes and replacing the voting functions in the decision-making process by the corresponding decoder functions. Our protocols can be seen as a recursive application of the information dispersal method for transmission of data described in [Rabi89]. Although our protocols are not early-stopping, in a number of relevant cases, the required amount of data communication is also considerably less than that needed in early-stopping protocols, like the early-stopping protocol for variable-length messages, described in Theorem 6 in [DoSt83]. We will refer to this protocol as the Dolev-protocol.

This new class of authenticated BAPs based on erasure-correcting codes will be defined and proved on basis of a class of algorithms which we call the Authenticated

Dispersed Joined Communication (ADJC) algorithms. These algorithms satisfy more liberal properties than those which are required for the BAPs. The ADJC-algorithms have strong similarities with the Dispersed Joined Communication (DJC) algorithms, which are designed for non-authenticated BAPs. The DJC-algorithms are described in [Krol91, Krol95].

3.2.1. The Authenticated Dispersed Joined Communication algorithms

In this section, we will describe the Authenticated Dispersed Joined Communication (ADJC) algorithms. These algorithms consist of a multicast process and a decision-making process, which are executed in a fully-connected lock-step synchronous system. It is assumed that all processors know the start of the algorithm and start the algorithm simultaneously.

In the multicast process, in a number of communication rounds, a message is transmitted from a source processor to a set of destination processors in the presence of a number of maliciously behaving processors. In the decision-making process, the destination processors calculate a decision about what the source has sent on basis of the information they received during the communication rounds.

It is assumed that a perfect communication link is available between every pair of correct processors in the system. This assumption is justified by the fact that we can model a link failure as a failure of one of its adjacent processors [SiLL90]. In the ADJC-algorithms, a link failure between a sending and a receiving processor will be modeled as a failure of the sending processor.

We assume furthermore that the ADJC-algorithms are designed in such a way that in each processor, every message for that processor will arrive in that processor in a certain predefined time slot. This is possible, because ADJC-algorithms are executed in a lock-step synchronous system, in which all processors know the start of the algorithm and start the algorithm simultaneously. This assumption makes it possible to deduce the path (i.e., the sequence of processors) along which a particular message *m* travelled from the source to a particular destination processor *c*, from the time slot in which message *m* arrives in *c*.

The ADJC-algorithms are defined such that they satisfy the following behavioural properties:

- *If the source and destination processor both function correctly, then the decision calculated during the decision-making process in the destination processor equals the original message in the source*
- *For an algorithm based on K rounds of information exchange in which a message is communicated from a source processor to a number of destination processors, it holds that if the decisions calculated in two well-functioning destinations differ from each other, then the number of maliciously behaving processors in the system is at least K.*

In Theorem 3.3. in Section 3.2.2., we will prove that the interactive consistency conditions are implied by these properties, provided that the number of maliciously behaving processors in the system is fewer than the number of communication rounds.

In order to be able to tolerate a number of maliciously behaving processors, the communication between the source and the destinations is *dispersed*, i.e., the message which needs to be transmitted is encoded into symbols of an erasure-correcting code, and these symbols are sent via different paths from the source to the destinations. Furthermore, the communication for different destinations is *joined* as much as possible, i.e., all paths from the source to these destinations are shared as much as is compatible with the additional requirement that a message is never relayed to a processor that it has already passed.

3.2.1.1. Definitions

The ADJC-algorithms are defined on a set *N* of fully interconnected processors. In a particular ADJC-algorithm, a particular message is sent in *K* communication rounds from a particular source processor *a* to all processors in the set *N*. See Figure 3.2. In general, many ADJC-algorithms with the same properties exist.

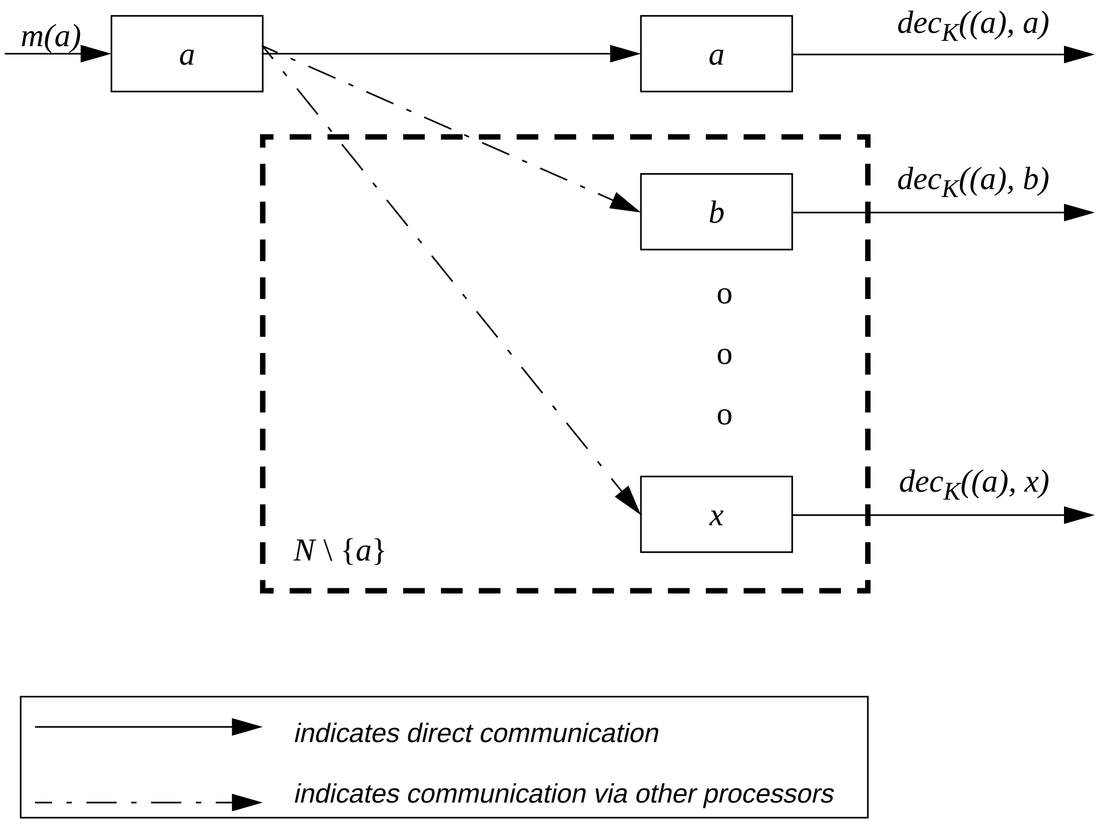


Figure 3.2. An ADJC-algorithm in the class A(T,K,a,N)

Therefore, we define classes

$$A\,(T,K,a,N\,) \tag{1}$$

of ADJC-algorithms in which:

T is the maximum number of maliciously behaving processors which is tolerated

K	is the number of communication rounds
a	is the source processor of the algorithm
N	is the set of processors in the system.

Obviously, these classes $A\ (T,K,a,N\ )$ of ADJC-algorithms are only defined if

$$K \geq 1 \text{ and } a \in N \tag{2a}$$

In order to exclude some pathological classes, we additionally require

$$|\,N\,| \geq K + 1 \tag{2b}$$

An algorithm in the class $A\ (\,T,K,a,N\ )$ forwards the original message in the source processor in K communication rounds to all processors in N. The original message in the source a is denoted by

$$m(\underline{\boldsymbol{p}};a) \text{ or by } m(a) \tag{3}$$

The prefix ($\underline{\boldsymbol{p}}$) to the source processor identifier a is used only when we need to distinguish between different messages in the same processor a. In that case, it denotes the path along which the message travelled to processor a.

For paths, we use the following notation:

N3.1.	( )	denotes the empty path
	(a)	denotes a path consisting of processor a
	($\underline{\boldsymbol{p}};a$)	denotes a path consisting of path ($\underline{\boldsymbol{p}}$) concatenated with processor a.

E.g., the path ($\underline{\boldsymbol{p}};a$) describes the path that follows all processors in path ($\underline{\boldsymbol{p}}$) (in order from left to right) and ends in processor a.

If a message $m(\underline{\boldsymbol{p}};a)$ (or $m(a)$) is sent to the processors in N by means of an algorithm from the class $A\ (T,K,a,N\ )$, then the results calculated in the processors d (with $d \in N$) are denoted by

$$dec_K\,((\underline{\boldsymbol{p}};a),d) \text{ (or by } dec_K\,((a),d)\text{)} \tag{4}$$

We will define $\mathbf{B}(a)$ as the next-set of processor a, i.e., the set of processors to which the message $m(a)$ from processor a is sent in the first communication round of the multicast process of the algorithm $A(\ T,K,\ a,\ N\ )$.

Definition of $K_{(a)}$ and $K_{(a)}^{(-1)}$.

In the ADJC-algorithms, for authentication purposes, for every processor $a \in N$, two functions $K_{(a)}$ and $K_{(a)}^{(-1)}$ (which are each other's inverses) are defined in such a way, that it is computationally infeasible to calculate $K_{(a)}$ from $K_{(a)}^{(-1)}$. Processor a reveals $K_{(a)}^{(-1)}$, but keeps $K_{(a)}$ secret [12].

12. An asymmetric cryptosystem, like, e.g., RSA [RiSA78], can be applied to implement the functions $K_{(a)}^{(-1)}$ and $K_{(a)}$. Function $K_{(a)}$ is chosen to be the secret cryptographic function D_a in [RiSA78], whereas $K_{(a)}^{(-1)}$ is chosen to be the corresponding publicly known cryptographic function E_a in [RiSA78]. More details about asymmetric cryptosystems and cryptographic functions can be found in Section 4.1.2.1.

Processor a is able to authenticate any message by appending a ***message digest*** to it, encrypted with $K_{(a)}$. Informally, a message digest (or ***hash value***) of a message is a fixed-length short sequence of bits, which is representative for the entire message. The message digest is obtained by applying a so-called ***message digest function*** on the message. The message digest function *digest* should be a cryptographic function known by all correct processors, with the property that it is computationally infeasible for any processor $p \in N$, which knows a certain (arbitrary) message m and its message digest $digest(m)$, to generate a message $m' \neq m$, such that $digest(m') = digest(m)$. In other words, function *digest* should be a ***strong one-way hash function*** [Merk82,Merk89]. See Section 4.1.2.3. for details. Most known message digest functions aim to satisfy this property. For more details about message digest functions, see, e.g., [Mull93].

The form of correct messages.

Messages transmitted in the ADJC-algorithms consist of two parts: an information part and a message digest, which is a function of the information part. For authentication purposes, the message digest is encrypted with the sender's secret cryptographic function.

A message $m(\underline{\mathbf{p}};a;b)$ which travelled along path $(\underline{\mathbf{p}})$ to a ($a \in N$), and is sent from a to b ($b \in \mathbf{B}(a)$) has the form

$$< i_{(\underline{\mathbf{p}};a;b)}, K_{(a)}\{digest(i_{(\underline{\mathbf{p}};a;b)})\} >,$$

in which

$i_{(\underline{\mathbf{p}};a;b)}$	is the information part of the message sent from a to b and
$K_{(a)}\{digest(i_{(\underline{\mathbf{p}};a;b)})\}$	is the message digest $digest(i_{(\underline{\mathbf{p}};a;b)})$, which is a function of $i_{(\underline{\mathbf{p}};a;b)}$ and which is encrypted with a's secret cryptographic function $K_{(a)}$.

For an empty path $(\underline{\mathbf{p}})$, we define $m((\underline{\mathbf{p}};a))$ as the original message in a.

Definition of $Z_{(a)}$ and $Z_{(a)}^{(-1)}$.

Message $m(\underline{\mathbf{p}};a)$ which is received in processor a along path $(\underline{\mathbf{p}})$ is related to the messages $m(\underline{\mathbf{p}};a;b)$ which a sends to all $b \in \mathbf{B}(a)$ in such a way, that $m(\underline{\mathbf{p}};a)$ is encoded into $|\mathbf{B}(a)|$ symbols of a ***T-erasure-correcting code***[13] $Z_{(a)}$ such that:

$$i_{(\underline{\mathbf{p}};a;b)} = Z_{(a)}(b)(m(\underline{\mathbf{p}};a)) \text{ for all } b \in \mathbf{B}(a)$$

Here, $Z_{(a)}(b)$ is the partial encoder function of $Z_{(a)}$ which delivers the symbol that has to be sent to b.

Let $Z_{(a)}$ be the encoder function of a T-erasure-correcting code (and $|\mathbf{B}(a)| > T$). Then, for a set $S_{(\underline{\mathbf{p}};a)}$ of $|\mathbf{B}(a)|$ information parts $i_{(\underline{\mathbf{p}};a;b)}$ (for $b \in \mathbf{B}(a)$) of which for at most T

13. A ***T-erasure correcting code*** is an erasure-correcting code that is capable of correcting up to T erasures.

information parts holds $i_{(\underline{\mathbf{p}};a;b)}$ = NIL, and of which all other information parts satisfy $i_{(\underline{\mathbf{p}};a;b)} = Z_{(a)}(b)(m(\underline{\mathbf{p}};a))$, the decoder function $Z_{(a)}^{(-1)}$ is defined by

$$Z_{(a)}^{(-1)}(S_{(\underline{\mathbf{p}};a)}) = m(\underline{\mathbf{p}};a)$$

For other sets S, we define $Z_{(a)}^{(-1)}(S)$ = NIL.

Definition of the functions $Y_{(a)}$ and $Y_{(a)}^{(-1)}$.
The authentication function $Y_{(a)}$ appends an encrypted message digest to an information part. Thus, for all information parts i:

$$Y_{(a)}(i) = <i,K_{(a)}\{digest(i)\}>$$

i.e Applying $Y_{(a)}$ on i results in applying function *digest* on i, by which message digest *digest*(i) is obtained, which is encrypted with a's secret cryptographic function $K_{(a)}$ and appended to i.

Any correct message m consists of an information part and a corresponding encrypted message digest and can be denoted by $m = <i,c>$, for some information part i and some encrypted message digest c, for which it holds that $K_{(a)}\{digest(i)\}=c$.

For any message m, function $Y_{(a)}^{(-1)}$ is defined by

$$Y_{(a)}^{(-1)}(m) = i, \text{ iff } m = <i,c> \text{ and } K_{(a)}\{digest(i)\}=c$$
$$Y_{(a)}^{(-1)}(m) = \text{NIL otherwise}$$

i.e. $Y_{(a)}^{(-1)}$) verifies if message m is mutilated by checking if m consists of an information part i and encrypted message digest c, such that c is compatible with i. If the result of applying functions *digest* and $K_{(a)}$ to i is equal to c then the message is not mutilated and i is returned, else $Y_{(a)}^{(-1)}$ returns NIL.

3.2.1.2. Construction of ADJC-algorithms

Assume a system consists of a set N of fully interconnected processors. At most $Z_{(T,N)}$ of these processors are allowed to behave maliciously. For any $T \geq 0$, and any non-empty set of processors **S**, $Z_{(T,\mathbf{S})}$ is defined by $Z_{(T,\mathbf{S})} = min(T, (|\mathbf{S}| - 1))$ (i.e., the minimum of T and (|**S**| - 1)).

The algorithms in the class $A(Z_{(T,N)},K,a,N)$ will be defined recursively with respect to K. Here, K is the number of communication rounds, and a denotes the source processor of the algorithm. The basis of the recursion is the case $K = 1$.

The construction of the algorithms in the class $A(Z_{(T,N)},1,a,N)$.
The multicast process of an algorithm in the class $A(Z_{(T,N)},1,a,N)$ consists of only one communication round. Let the original message in a be $m(\underline{\mathbf{p}};a)$.

During communication round 1 of the multicast process, processor a sends the original message directly and unchanged to all processors in $(N \setminus \{a\})$. Processor a also keeps a copy of $m(\underline{\mathbf{p}};a)$ itself in order to use it in the decision-making process.

Then, the decision-making process is executed in which in each processor $d \in (N \setminus \{a\})$, the message $m(\underline{\mathbf{p}};a;d)$ (received from a) is taken as decision $dec_1((\underline{\mathbf{p}};a),d)$. The decision $dec_1((\underline{\mathbf{p}};a),a)$ in processor a is equal to the stored message $m(\underline{\mathbf{p}};a)$. Figure 3.3. illustrates an ADJC-algorithm in the class $A(Z_{(T,N)},1,a,N)$.

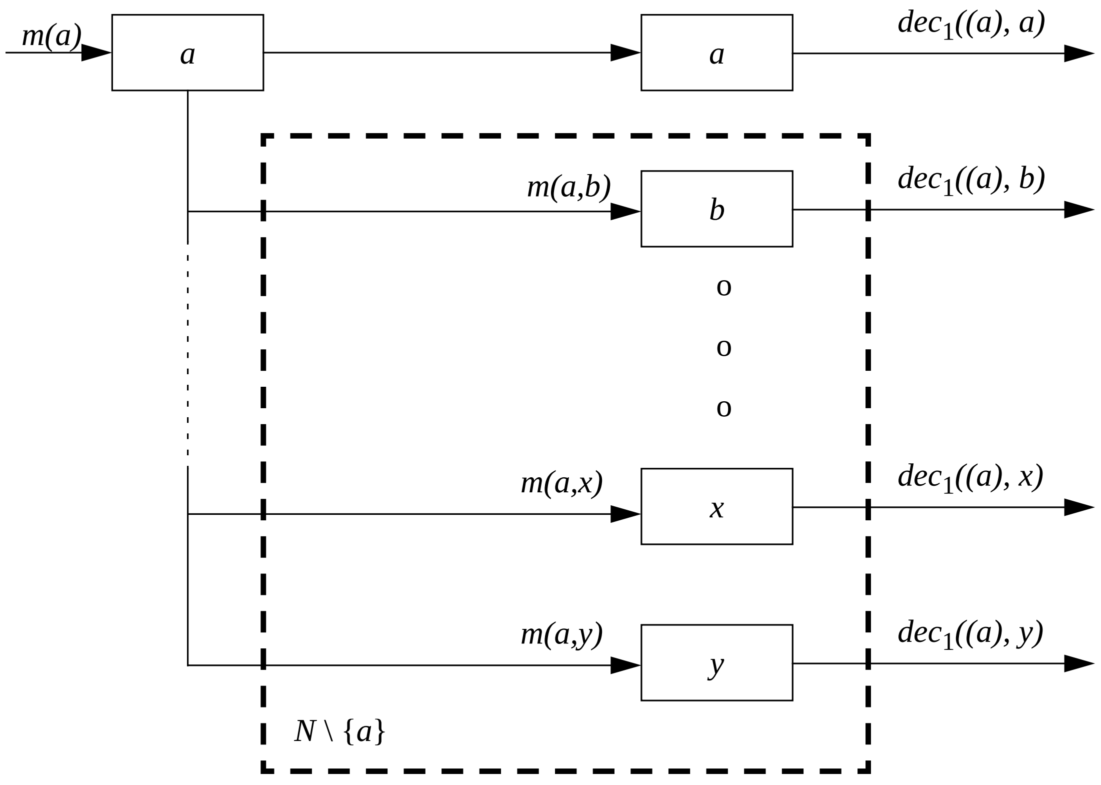


Figure 3.3. An ADJC-algorithm in the class $A(Z_{(T,N)},1,a,N)$

The behavioural aspects of the algorithms in the class $A(Z_{(T,N)},1,a,N)$ (starting from correctly functioning processors) are defined by

$$d \in (N \setminus \{a\}) \Rightarrow m(\underline{\mathbf{p}};a;d) = m(\underline{\mathbf{p}},a)$$
$$d \in (N \setminus \{a\}) \Rightarrow dec_1((\underline{\mathbf{p}};a),d) = m(\underline{\mathbf{p}};a;d)$$
$$dec_1((\underline{\mathbf{p}};a),a) = m(\underline{\mathbf{p}};a)$$

The recursive construction of the algorithms in the class $A(Z_{(T,N)},K,a,N)$ with $K > 1$ in terms of algorithms from the set of classes $A(Z_{(T,N \setminus \{a\})},K-1,b,N \setminus \{a\})$ with $b \in N \setminus \{a\}$.

Let the original message in a be $m(\underline{\mathbf{p}};a)$. The construction of the algorithms in the class

$A(Z_{(T,N)},K,a,N)$ with $K > 1$ is based on applying the partial encoder functions $Z_{(a)}(b)$ (for every $b \in \mathbf{B}(a)$ where $\mathbf{B}(a) \subset (N \setminus \{a\})$) of some $Z_{(T,N \setminus \{a\})}$-erasure-correcting code $Z_{(a)}$ on message $m(\underline{\mathbf{p}};a)$ (here, $Z_{(T,N \setminus \{a\})} = min(T,(|N \setminus \{a\}| - 1))$). This results in the encoding of message $m(\underline{\mathbf{p}};a)$ into symbols of a $Z_{(T,N \setminus \{a\})}$-erasure-correcting code. These symbols are denoted by $i_{(\underline{\mathbf{p}};a;b)}$. By applying authentication function $Y_{(a)}$ to each symbol $i_{(\underline{\mathbf{p}};a;b)}$, the messages $m(\underline{\mathbf{p}};a;b)$ are obtained which are transmitted to the processors $b \in \mathbf{B}(a)$. Every processor b forwards its message $m(\underline{\mathbf{p}};a;b)$ to the destinations by means of an algorithm from the class $A(Z_{(T,N \setminus \{a\})},K - 1,b,N \setminus \{a\})$.

Figure 3.4. illustrates an ADJC-algorithm in the class $A(Z_{(T,N)},K,a,N)$ with $K > 1$.

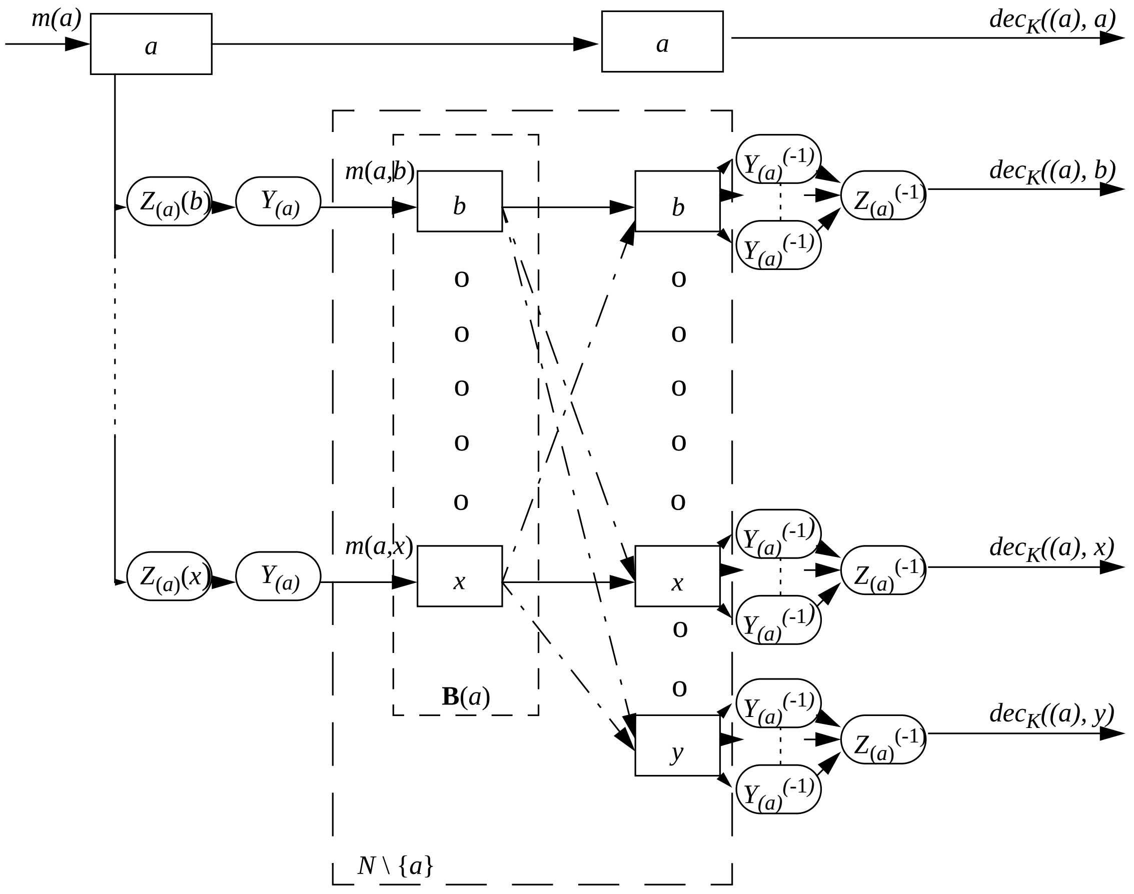


Figure 3.4. An ADJC-algorithm in the class $A(Z_{(T,N)},K,a,N)$ with $K > 1$

The ADJC-algorithms in the class $A(Z_{(T,N)},K,a,N)$ with $K > 1$ are constructed as follows:

(1) During communication round 1

(a) Let the original message in a be $m(\underline{\mathbf{p}};a)$. This message $m(\underline{\mathbf{p}};a)$ is kept stored in a in order to be used later on during round K in the decision-making process.

(b) Furthermore, in the source processor a, for every $b \in \mathbf{B}(a)$ a partially encoded and authenticated version $m(\underline{\mathbf{p}};a;b)$ of the original message $m(\underline{\mathbf{p}};a)$ is calculated such that $m(\underline{\mathbf{p}};a;b) = Y_{(a)}(Z_{(a)}(b)(m(\underline{\mathbf{p}};a)))$.

(c) Each processor $b \in \mathbf{B}(a)$ receives from a a message $m(\underline{\mathbf{p}};a;b)$. The next-set $\mathbf{B}(a)$ must contain at least $Z_{(T,N \setminus \{a\})} + 1$ processors.

(2) During the communication rounds 2, … K, each processor b in the next-set $\mathbf{B}(a)$ forwards the received message $m(\underline{\mathbf{p}};a;b)$ to the destinations indicated by $(N \setminus \{a\})$ by means of an algorithm from the class $A(\, Z_{(T,N \setminus \{a\})},K - 1,b,N \setminus \{a\})$. The results of these algorithms in the destinations $d \in (N \setminus \{a\})$ are denoted by $dec_{K\text{-}1}((\underline{\mathbf{p}};a;b),d)$. These results are calculated during the first part of the decision-making process.

(3) During the second part of the decision-making process, the decision $dec_K((\underline{\mathbf{p}};a),d)$ in the processors d with $d \in (N \setminus \{a\})$ is obtained as follows. First, function $Y_{(a)}^{(-1)}$ is applied on the results $dec_{K\text{-}1}\,((\underline{\mathbf{p}};a;b),d)$ with $b \in \mathbf{B}(a)$. The resulting information parts $i_{(\underline{\mathbf{p}};a;b)}$ form the symbols of a $Z_{(T,N \setminus \{a\})}$-erasure-correcting code $Z_{(a)}$. The result of applying the corresponding decoder function $Z_{(a)}^{(-1)}$ on this set of information parts is taken as decision $dec_K((\underline{\mathbf{p}};a),d)$. The decision $dec_K((\underline{\mathbf{p}};a),a)$ is obtained by taking the message value $m(\underline{\mathbf{p}};a)$ which had been kept stored in processor a. How this decision is calculated is illustrated in Figure 3.5. and 3.6.

The behavioural aspects of the algorithms in the class $A(\, Z_{(T,N\,)},K,a,N\,)$ with $K > 1$ are defined by

$$m(\underline{\mathbf{p}};a;b) = Y_{(a)}(Z_{(a)}(b)(m(\underline{\mathbf{p}};a))) \text{ for all } b \in \mathbf{B}(a)$$

$$dec_{K\text{-}1}((\underline{\mathbf{p}};a;b),d) \text{ follows from } m(\underline{\mathbf{p}};a;b) \text{ based on an algorithm from the class } A(\, Z_{(T,N \setminus \{a\})},K - 1,b,N \setminus \{a\})$$

$$d \neq a \Rightarrow dec_K((\underline{\mathbf{p}};a),d) = Z_{(a)}^{(-1)} \text{ applied on the values } Y_{(a)}^{(-1)}(dec_{K\text{-}1}((\underline{\mathbf{p}};a;b),d)) \text{ with } b \in \mathbf{B}(a)$$

$$d = a \Rightarrow dec_K((\underline{\mathbf{p}};a),d) = m(\underline{\mathbf{p}};a)$$

3.2.1.3.The existence of ADJC-algorithms in the classes $A\,(Z_{(T,N\,)},K,a,N\,)$

The construction of ADJC-algorithms is possible if and only if

- A next-set $\mathbf{B}(a)$ can be found which is a non-empty subset of $(N \setminus \{a\})$ and which contains at least $Z_{(\mathrm{T},N \setminus \{a\})} + 1$ processors.
- A $Z_{(T,N \setminus \{a\})}$-erasure-correcting code exists of which the code words consist of $|\mathbf{B}(a)|$ symbols.
- The classes $A\,(Z_{(T,N \setminus \{a\})},K - 1,b,N \setminus \{a\})$ of ADJC-algorithms with $b \in \mathbf{B}(a)$ are all non-empty. (5)

We will now investigate for which parameters the classes $A\,(Z_{(T,N\,)},K,a,N\,)$ of ADJC

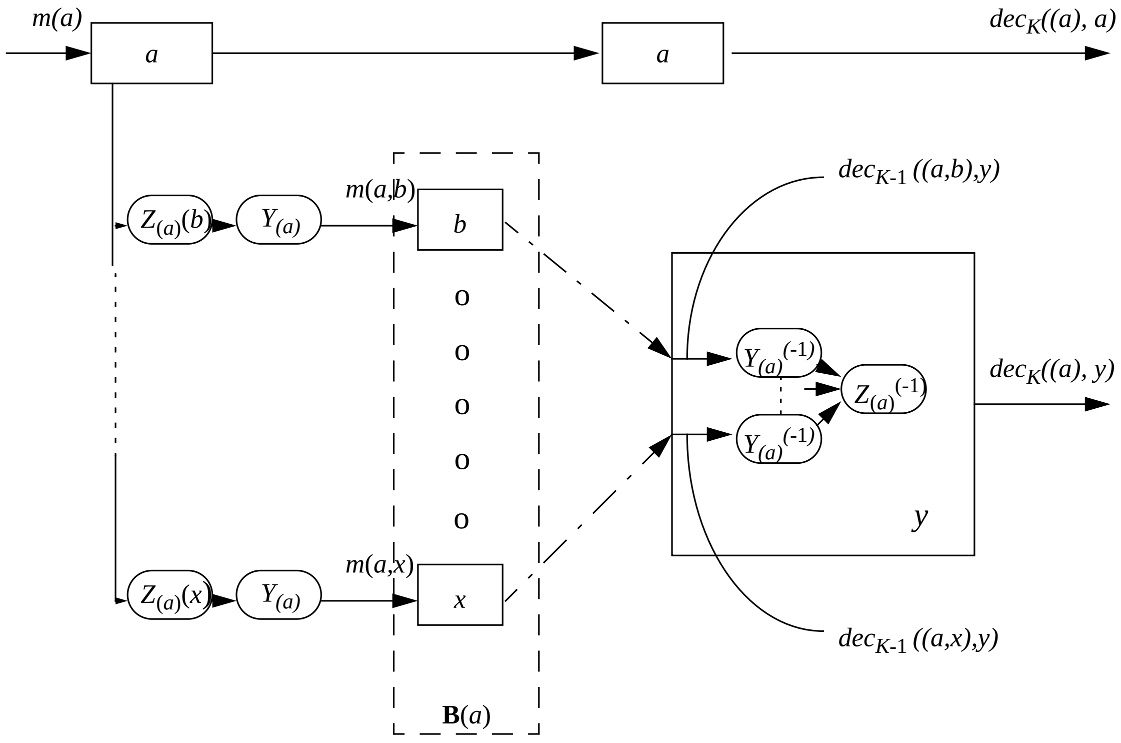


Figure 3.5. *Calculation of $dec_K((a),y)$ in a processor $y \notin$ **B**(a) from the partially encoded messages sent from processor a to the processors in* **B**(a)

algorithms are non-empty. Recall from (2a) and (2b) that the classes $A\ (Z_{(T,N\ )},K,a,N\ )$ of ADJC algorithms are only defined if

$$K \geq 1 \text{ and } a \in N \text{ and } |\, N\, | \geq K + 1 \qquad (6)$$

If these so-called general constraints of class $A\ (Z_{(T,N\ )},K,a,N\ )$ are not satisfied, then this class is empty by definition.

Within this context, the next theorem will show that the non-emptiness of the classes $A\ (Z_{(T,N\ )},K,a,N\ )$ only depends on the number of processors in the system $|\, N\, |$ and the number of communication rounds K.

THEOREM 3.1.

- For all T with $T \geq 0$, and
- for all K with $K \geq 1$, and
- for all fully interconnected systems consisting of $|\, N\, |$ processors, with $|\, N\, | \geq K + 1$, and
- for all source processors $a \in N$,

the class of algorithms $A\ (Z_{(T,N\ )},K,a,N\ )$ is non-empty if and only if

$$|\, N\, | \geq K + 1$$ ❏

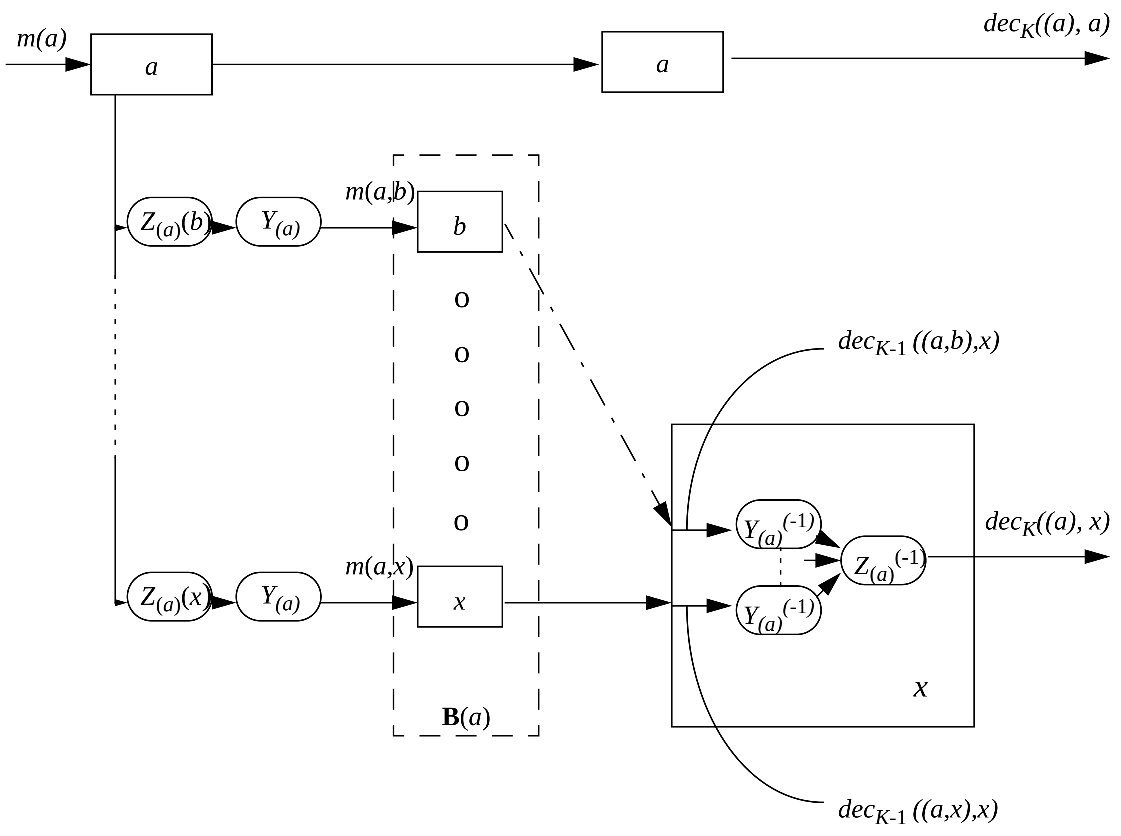


Figure 3.6. *Calculation of $dec_K((a),x)$ in a processor $x \in$ **B**(a) from the partially encoded messages sent from processor a to the processors in* **B***(a)*

Proof:

Theorem 3.1. will be proved by induction on *K*. The basis of the induction is $K = 1$. Throughout the proof, we assume that the parameters *K* and *T*, the source processor *a*, and set *N* satisfy

$$T \geq 0 \text{ and } K \geq 1 \text{ and } a \in N \text{ and } |N| \geq K + 1 \tag{7}$$

Basis: *K* = 1.

The class of algorithms $A\,(Z_{(T,N\,)},1,a,N\,)$ is non-empty if and only if the general constraints as stated in (7) are fulfilled, i.e.:

$$T \geq 0 \text{ and } a \in N \text{ and } |N| \geq 2 \tag{8}$$

From this immediately follows the necessary and sufficient requirement

$$|N| \geq 2 \tag{9}$$

This proves Theorem 3.1. for $K = 1$.

Induction step: $K > 1$.
Suppose Theorem 3.1. holds for $K - 1$ with $K \geq 2$. So suppose for all T, a, and N, with $T \geq 0$ and $a \in N$ and $|N| \geq K$ holds that the class of algorithms $A(Z_{(T,N)}, K-1, a, N)$ is non-empty if and only if

$$|N| \geq K \tag{10}$$

From (5), we recall that the construction of algorithms based on K rounds of information exchange is possible if and only if

- A next-set $\mathbf{B}(a)$ can be found which is a non-empty subset of $(N \setminus \{a\})$ and which contains at least $Z_{(T,N \setminus \{a\})} + 1$ processors.
- A $Z_{(T,N \setminus \{a\})}$-erasure-correcting code exists of which the code words consist of $|\mathbf{B}(a)|$ symbols.
- The classes $A(Z_{(T,N \setminus \{a\})}, K-1, b, N \setminus \{a\})$ of ADJC algorithms with $b \in \mathbf{B}(a)$ are all non-empty.

The first requirement can be satisfied if and only if

$$|N \setminus \{a\}| \geq 1 \tag{11}$$

A necessary and sufficient requirement for predicate (11) is

$$|N| \geq 2 \tag{12}$$

The second requirement can always be fulfilled since $|\mathbf{B}(a)| \geq Z_{(T,N \setminus \{a\})} + 1$ and because it is always possible to construct a $Z_{(T,N \setminus \{a\})}$-erasure-correcting code if the code words contain more than $Z_{(T,N \setminus \{a\})}$ symbols [VaOo89].

The third requirement is fulfilled if the general constraints (6) of the classes
$A(Z_{(T,N \setminus \{a\})}, K-1, b, N \setminus \{a\})$ are satisfied, i.e.:

$$K - 1 \geq 1 \text{ and } b \in (N \setminus \{a\}) \tag{13}$$

and

$$|(N \setminus \{a\})| \geq K \tag{14}$$

and if and only if the condition expressed in (10) is satisfied, i.e.:

$$|(N \setminus \{a\})| \geq K \tag{15}$$

Predicate (13) is satisfied by (7), the assumption $K > 1$, and the facts that $b \in \mathbf{B}(a)$ and $\mathbf{B}(a) \subset (N \setminus \{a\})$ (otherwise the construction of the ADJC algorithms is not possible).

Predicate (14) and (15) are satisfied if and only if

$$|N| \geq K + 1 \tag{16}$$

Hence, assumption (10) implies that for all T, a, N, with $T \geq 0$ and $a \in N$ and $|N| \geq K + 1$ holds that the class of algorithms $A(Z_{(T,N)}, K, a, N)$ is non-empty if and only if

$$|N| \geq K + 1 \tag{17}$$

Thus Theorem 3.1. holds for any $K \geq 1$. ❑

3.2.1.4.Behavioural properties of the ADJC-algorithms in the presence of at most $Z_{(T,N)}$ maliciously behaving processors

The ADJC-algorithms satisfy the following behavioural properties:

THEOREM 3.2.
Let N be a set of fully interconnected processors. At most $Z_{(T,N)}$ of these processors behave maliciously. Suppose that by means of any ADJC-algorithm from the class A ($Z_{(T,N)}$,K,a,N) a message $m(a)$ is transmitted from the source processor a to all processors in set N. Let the decisions calculated in the destinations d with $d \in N$ be denoted by $dec_K((a),d)$, then

- 1. If the source processor a and a destination d both function correctly, then the result $dec_K((a),d)$ of the algorithm calculated in d equals the original message $m(a)$ in a.
- 2. For any two well-functioning destinations d and e, it holds that if the results $dec_K((a),d)$ and $dec_K((a),e)$ are unequal, then the number of maliciously behaving processors in the system is at least K. ❑

Proof:
We first prove property 1.

Assume that a and d are two well-functioning processors in the system ($a,d \in N$) in which a is the source processor of the algorithm, and d is one of the destination processors.

If $a = d$, then, according to the construction of the algorithms of class A ($Z_{(T,N)}$,K,a,N) described in Section 3.2.1.2., it holds that $dec_K((a),d) = m(a)$.

So we only need to consider the case $d \neq a$, i.e., $d \in (N \setminus \{a\})$. For these cases we prove $dec_K((a),d) = m(a)$ by induction on K.

Basis: *K* = 1.
Algorithms in the class A ($Z_{(T,N)}$,1,a,N) send the message $m(a)$ directly and unchanged to the destination d, and the decision calculated in d equals the message received from a. So, because we assume that a and d function correctly, it holds that $dec_1((a),d) = m(a)$.

Induction step: *K* > 1.
The algorithms in the class A ($Z_{(T,N)}$,K,a,N) have been constructed from the algorithms in the class A ($Z_{(T,N \setminus \{a\})}$,$K$ - 1,b,$N \setminus \{a\}$) with $b \in \mathbf{B}(a)$ and $\mathbf{B}(a) \subset (N \setminus \{a\})$.

From the latter algorithms, we know from the induction hypothesis that if a message $m(a;b)$ is communicated by a well-functioning source processor b to a well-functioning destination processor d in ($N \setminus \{a\}$), then

$$dec_{K-1}((a;b),d) = m(a;b) \qquad (18)$$

From the construction we know the following:

In processor a, for all $b \in \mathbf{B}(a)$, by means of the partial encoder functions $Z_{(a)}(b)$, the original message is encoded into $|\mathbf{B}(a)|$ symbols. By applying $Y_{(a)}$ to such a symbol, a message digest is appended to it, resulting in a message $m(a;b)$. During communication round 1, every processor $b \in \mathbf{B}(a)$ receives a message $m(a;b)$ where:

$$m(a;b) = Y_{(a)}(Z_{(a)}(b)(m(a))) \tag{19}$$

Each of these processors b forwards the message $m(a;b)$ to the destinations in $(N \setminus \{a\})$ by means of an algorithm from the class $A\,(Z_{(T,N \setminus \{a\})}, K - 1, b, N \setminus \{a\})$. As the result of these algorithms, in each destination d of the set $(N \setminus \{a\})$, $|\mathbf{B}(a)|$ decisions $dec_{K-1}((a;b),d)$ will become available. If processor b functions correctly, then, according to (18) and (19), it holds that

$$dec_{K-1}((a;b),d) = Y_{(a)}(Z_{(a)}(b)(m(a))) \tag{20}$$

In each processor d, function $Y_{(a)}^{(-1)}$ is applied to these decisions $dec_{K-1}((a;b),d)$ with $b \in \mathbf{B}(a)$.

From the construction, we know that the next-set $\mathbf{B}(a)$ for algorithms in the class $A\,(Z_{(T,N)}, K, a, N)$ contains at least $min((T + 1), (\,|N \setminus \{a\}|\,))$ processors.

Now, we can distinguish two cases:
1. $|\,N \setminus \{a\}\,| \geq T + 1$
2. $|\,N \setminus \{a\}\,| < T + 1$

Case 1: $|\,N \setminus \{a\}\,| \geq T + 1$
Since $|\,N \setminus \{a\}\,| \geq T + 1$, the next-set $\mathbf{B}(a)$ contains at least $T + 1$ processors. So, $\mathbf{B}(a)$ contains at least one well-functioning processor, and thus, processor d contains at least one decision which satisfies (20).

Case 2: $|\,N \setminus \{a\}\,| < T + 1$
In this case, the next-set $\mathbf{B}(a)$ contains $|\,N \setminus \{a\}\,|$ processors. Therefore, since $\mathbf{B}(a) \subset (N \setminus \{a\})$, we must conclude that $\mathbf{B}(a) = (N \setminus \{a\})$. Because $d \in (N \setminus \{a\})$, and we assumed that d is a well-functioning processor, $\mathbf{B}(a)$ contains at least one well-functioning processor (viz. processor d), and thus, processor d contains at least one decision which satisfies (20).

So, in both cases, processor d contains at least one decision which satisfies (20). From the foregoing, we see that at most $Z_{(T,N \setminus \{a\})}$ processors b behave maliciously (Recall that $Z_{(T,N \setminus \{a\})} = min(T, (|\,N \setminus \{a\}| - 1)))$. So at most $Z_{(T,N \setminus \{a\})}$ decisions do not satisfy (20). The code $Z_{(a)}$ is $Z_{(T,N \setminus \{a\})}$-erasure-correcting, and thus, the result of applying decoder function $Z_{(a)}^{(-1)}$ on the values $Y_{(a)}^{(-1)}(dec_{K-1}((a;b),d))$ (which is taken as final decision $dec_K((a),d)$) must be equal to $m(a)$. This completes the proof of property 1.

Now we prove property 2.

Assume that two destinations d and e (with $d,e \in N$) behave correctly and that $dec_K((a),d) \neq dec_K((a),e)$.

If the source processor a functions correctly then, by property 1, it holds that $dec_K((a),d) = m(a) = dec_K((a),e)$ conflicting the assumption $dec_K((a),d) \neq dec_K((a),e)$. Thus, processor a behaves maliciously and thus (since we assumed that d and e function correctly), $d \neq a$ and $e \neq a$. The proof is again by induction on K.

Basis: $K = 1$
We already concluded that processor a behaves maliciously. Hence, the system contains at least one maliciously behaving processor.

Induction step: $K > 1$
Recall that during communication round 1 the symbols are sent by processor a to the processors b with $b \in \mathbf{B}(a)$. During the communication rounds 2, …K, each of these symbols $m(a;b)$ is forwarded to the destinations in the set $(N \setminus \{a\})$ by means of an algorithm from the class $A(Z_{(T,N \setminus \{a\})}, K\text{-}1,b,N \setminus \{a\})$. The results of these algorithms calculated in the processors d with $d \in (N \setminus \{a\})$ are denoted by $dec_{K\text{-}1}((a;b),d)$.

Let $dec_{K\text{-}1}((a;b),d)$ and $dec_{K\text{-}1}((a;b),e)$ with $b \in \mathbf{B}(a)$ be decisions calculated in processors d and e. We already concluded that processor a behaves maliciously and that $d \neq a$ and $e \neq a$. Hence, $d,e \in (N \setminus \{a\})$. Since $d \neq a$ and $e \neq a$, the decision $dec_K((a),d)$ is based on applying the functions $Y_{(a)}^{(-1)}$ and $Z_{(a)}^{(-1)}$ on the decisions $dec_{K\text{-}1}((a;b),d)$ with $b \in \mathbf{B}(a)$, whereas the decision $dec_K((a),e)$ is based on applying the same functions $Y_{(a)}^{(-1)}$ and $Z_{(a)}^{(-1)}$ on the decisions $dec_{K\text{-}1}((a;b),e)$ with $b \in \mathbf{B}(a)$. It follows that

$$(\forall\, b \in \mathbf{B}(a) :: dec_{K\text{-}1}((a;b),d) = dec_{K\text{-}1}((a;b),e)) \qquad (21)$$

would imply $dec_K((a),d) = dec_K((a),e)$. The latter, however, conflicts with the assumption $dec_K((a),d) \neq dec_K((a),e)$, so we conclude

$$(\exists\, b \in \mathbf{B}(a) :: dec_{K\text{-}1}((a;b),d) \neq dec_{K\text{-}1}((a;b),e)) \qquad (22)$$

Recall that our assumption implies $d,e \in (N \setminus \{a\})$. So from the definition of the construction of the algorithms in the class $A\,(Z_{(T,N\,)},K,a,N\,)$ we know that the decisions $dec_{K\text{-}1}((a;b),d)$ respectively $dec_{K\text{-}1}((a;b),e)$ are the result of algorithms from the classes $A\,(\,Z_{(T,N \setminus \{a\})},K\text{-}1,b,N \setminus \{a\})$ with $b \in \mathbf{B}(a)$.

According to the induction hypothesis, it holds for the latter classes that if the processors d and e both function correctly and $dec_{K\text{-}1}((a;b),d) \neq dec_{K\text{-}1}((a;b),e)$, then the number of maliciously behaving processors in the set $(N \setminus \{a\})$ must be at least K-1. And thus, with (22), we conclude that the set $(N \setminus \{a\})$ must contain at least K-1 maliciously behaving processors. We already concluded that processor a behaves maliciously. Hence, set N contains at least K maliciously behaving processors.

This completes the proof of Theorem 3.2. ❏

3.2.2. A class of Byzantine Agreement Protocols based on the ADJC-algorithms

The class of authenticated Byzantine Agreement Protocols based on erasure-correcting codes is a subclass of the ADJC-algorithms defined by:

$$A\ (T,K,a,N\ ) \text{ with } T \geq 0, \text{ and } K = T + 1 \tag{23}$$

From Theorem 3.1., we find that the class of protocols $A\ (T,T + 1,a,N\ )$ is non-empty if and only if $|\ N\ | \geq T + 2$.

THEOREM 3.3.
Any Byzantine Agreement Protocol from the class $A\ (T,T + 1,a,N\ )$, with $|\ N\ | \geq T + 2$ satisfies the interactive consistency conditions, stated formally as (cf. Section 3.1.3.):

IC1. $(\forall\ d \in C :: a \in C \Rightarrow dec_{T+1}((a),d) = m(a))$
IC2. $(\forall\ d,e \in C :: dec_{T+1}((a),d) = dec_{T+1}((a),e))$

in which C is any set of well-functioning processors such that $|\ C\ | \geq |\ N\ | - T$, and $C \subset N$ and $dec_{T+1}((a),d)$ (respectively $dec_{T+1}((a),e)$) with $d,e \in N$ denotes the decision in a processor d (respectively e) about what processor a tried to send. ❑

Proof:
Consider Byzantine Agreement Protocols which are defined by the non-empty class of ADJC-algorithms $A\ (T,T + 1,a,N\ )$ with $T \geq 0$, and $|\ N\ | \geq T + 2$.

Let C be any set of well-functioning processors such that $|\ C\ | \geq |\ N\ | - T$, and $C \subset N$.

From the first part of Theorem 3.2., we know that if the source processor a and a destination d both function correctly, then the result $dec_K((a),d)$ of the protocol calculated in d equals the original message $m(a)$ in a. Thus:

$$(\forall\ d \in C :: a \in C \Rightarrow dec_K\ ((a),d) = m(a))$$

This proves the first interactive consistency condition IC1.

From the second part of Theorem 3.2., we know that for any two well-functioning destinations d and e it holds that if the results $dec_K((a),d)$ and $dec_K((a),e)$ are unequal, then the number of maliciously behaving processors in the system is at least K.

However, this conflicts with the constraint $K = T + 1$ and the assumption that at most T processors behave maliciously. Thus, if both processors d and e behave correctly, the decisions $dec_K((a),d)$ and $dec_K((a),e)$ must be identical. So

$$(\forall\ d,e \in C :: dec_K\ ((a),d) = dec_K\ ((a),e)).$$

This completes the proof of Theorem 3.3. ❑

3.2.3. The construction of authenticated Byzantine Agreement Protocols based on erasure-correcting codes

Some freedom is left in the definition of the Byzantine Agreement Protocols based on erasure-correcting codes, starting from the ADJC-algorithms. A reduction in the number of messages that needs to be transmitted between the processors can be obtained in two ways, viz.:

- by minimizing the number of directions in which the messages are multicast each round. This is done in the so-called Minimum Direction protocols, described in Section 3.2.3.1.
- by maximizing the length of the applied erasure-correcting code. This is done in the so-called Maximum Coding protocols, described in Section 3.2.3.2.

From the construction we immediately see that maximizing the length of the error-correcting code causes an increase in the number of processors to which messages are sent. However, the size of the messages that is transmitted decreases with an increasing code length. In Section 3.2.4.6. we show that this decrease is more efficient than reducing the number of directions.

3.2.3.1. The Minimum Direction protocols

In this subclass of ADJC-algorithms, only repetition codes are applied. Thus, during each communication round a received message $m(\underline{\boldsymbol{p}})$ is multicast unchanged to a number of processors given by the set $\mathbf{B}(\underline{\boldsymbol{p}})$. During each round, except the last one, each message is relayed to exactly $Z_{(T,N \setminus \mathrm{set}(\underline{\boldsymbol{p}}))}$ processors. (For a path $(\underline{\boldsymbol{p}})$, set$(\underline{\boldsymbol{p}})$ is defined as the set of processors contained in path $(\underline{\boldsymbol{p}})$). During the last round a message $m(\underline{\boldsymbol{p}})$ is sent to the processors in $N \setminus \mathrm{set}(\underline{\boldsymbol{p}})$.

3.2.3.2. The Maximum Coding protocols

When a $Z_{(T,N \setminus \mathrm{set}(\underline{\boldsymbol{p}}))}$-erasure-correcting code $Z_{(\underline{\boldsymbol{p}})}$ is applied, consisting of code words of $n_{(\underline{\boldsymbol{p}})}$ symbols of size $b_{(\underline{\boldsymbol{p}})}$ and data words consisting of $k_{(\underline{\boldsymbol{p}})}$ symbols of the same size, then the total amount of information multicast by a processor during round t $(2 \le t \le T)$ is a factor $n_{(\underline{\boldsymbol{p}})}/k_{(\underline{\boldsymbol{p}})}$ more than the amount of information received by that processor during round t - 1. (Notice that in the last communication round, no encoding takes place). From [VaOo89], we know that a $Z_{(T,N \setminus \mathrm{set}(\underline{\boldsymbol{p}}))}$-erasure-correcting code $Z_{(\underline{\boldsymbol{p}})}$ can be constructed if and only if $n_{(\underline{\boldsymbol{p}})} \ge k_{(\underline{\boldsymbol{p}})}+Z_{(T,N \setminus \mathrm{set}(\underline{\boldsymbol{p}}))}$. In the Maximum Coding protocols, the fraction $n_{(\underline{\boldsymbol{p}})}/k_{(\underline{\boldsymbol{p}})}$ is kept as low as possible. This is achieved by maximizing $k_{(\underline{\boldsymbol{p}})}$, while keeping $n_{(\underline{\boldsymbol{p}})}$ as low as possible. We choose $n_{(\underline{\boldsymbol{p}})} = k_{(\underline{\boldsymbol{p}})}+Z_{(T,N \setminus \mathrm{set}(\underline{\boldsymbol{p}}))}$, and maximize $k_{(\underline{\boldsymbol{p}})}$, i.e., the partial encoding is spread across as many processors as is allowed. Codes with these parameters always exist if the symbol size $b_{(\underline{\boldsymbol{p}})}$ satisfies $b_{(\underline{\boldsymbol{p}})} \ge {}^2\log(n_{(\underline{\boldsymbol{p}})}\text{-}1)$ [Krol91]. In literature, these codes are known as Maximum Distance Separable (MDS) codes. We choose the set $\mathbf{B}(\underline{\boldsymbol{p}})$ to be as large as possible, thus

$$|\mathbf{B}(\underline{\boldsymbol{p}})| = |\ N \setminus \mathrm{set}(\underline{\boldsymbol{p}})\ | \text{ for } 1 \le |\ (\underline{\boldsymbol{p}})\ | \le T$$

In each round t (with $|\ (\underline{\boldsymbol{p}})\ | = t$ and $1 \le t \le T$) of a Maximum Coding protocol, a $Z_{(T,N \setminus \mathrm{set}(\underline{\boldsymbol{s}}))}$-erasure-correcting code is applied with parameters:

- *number of code word symbols:*

$$n(\underline{\boldsymbol{p}}) = N - t$$

- *number of data word symbols:*

$$k(\underline{\boldsymbol{p}}) = N - t - Z_{(T,N \setminus \mathrm{set}(\underline{\boldsymbol{p}}))}$$

- *symbol size in number of bits:*

  $b(\underline{\boldsymbol{p}}) \geq {}^{2}\log(N\text{-}t\text{-}2)$ if $N - t > T + 1$

  $b(\underline{\boldsymbol{p}}) \geq 1$ if $N - t \leq T + 1$

3.2.4. A comparison with existing BAPs

We will now compare our authenticated BAPs with some well-known lock-step synchronous deterministic authenticated BAPs, described in literature.

A first solution to the problem was given in [PeSL80] and described as the SM protocol in [LaSP82]. This protocol, which we will refer to as the Lamport-protocol, is inefficient with respect to the required amount of data communication. Therefore, in literature a number of more efficient protocols have been proposed. An extensive overview is given in [BaMD93]. In [DoRe85], a lower bound was given for the number of messages required, together with algorithms for 1-bit messages that meet this lower bound. For messages greater than 1 bit, a very efficient early-stopping protocol is given in Theorem 6 in [DoSt83]. We will refer to this protocol as the Dolev-protocol.

We will compare our BAPs with the SM protocol described in [LaSP82], referred to as the Lamport-protocol, and the protocol described in Theorem 6 in [DoSt83], which we will call the Dolev-protocol.

3.2.4.1. The criteria

The criteria by means of which the protocols will be compared are:

- The relative number of messages, *mess*, that needs to be transmitted between the processors in terms of the original message that is multicast by the source
- The minimum size, *msize*, of the original message.

3.2.4.2. The Lamport-protocol

In the Lamport-protocol, the required number of messages *mess* is:

$$mess = \sum_{i=1}^{T+1} i! \cdot \begin{bmatrix} N-1 \\ i \end{bmatrix}$$

We choose the amount of information of the original message that is multicast by the source be the unit. So the message size *msize* of all messages in the Lamport-protocol is 1.

3.2.4.3. The Dolev-protocol

For the Dolev-protocol, we calculate the maximum number of messages (*mess*) that is required. We distinguish between the case $N > 2T$+1 and the case $N \leq 2T$+1.

The case $N > 2T$+1.
In case $N > 2T$+1, the maximally required number of messages mess is given by the following formulas. If $T = 0$, only 1 round of communication is performed and thus the number of messages *mess*= N-1. If T=1, two communication rounds are necessary and the required number of messages *mess* = $(N - 1) + (2T+1)\cdot(N - 2) = 4N - 7$. If $T \geq 2$,

then *mess* = (N - 1) + (2T+1)·(N - 2) + (2T+1)·(N - 3).

The case T+2 ≤ N ≤ 2T+1.
In case T+2 ≤ N ≤ 2T+1, the maximally required number of messages *mess* is given by the following formulas. If T = 1, two communication rounds are necessary and the required number of messages *mess* = (N - 1) + (N - 1)·(N - 2) = 4 (since N = 3). If T ≥ 2, then *mess* = (N - 1) + (N - 1)·(N - 2) + (N - 1)·(N - 3).

The message size *msize* of all messages in the Dolev-protocol is 1.

3.2.4.4. The Minimum Direction protocol

If N > T+2, there is some design freedom in the design of a BAP.

In the Minimum Direction protocols, therefore, in each communication round, a message is relayed to a set of T+1 receivers that did not receive the message before. If N < 2T+1, then after N-T-1 rounds, such a set cannot be found anymore. In this case, the message is relayed to all T processors that did not receive the message before. In the next round, these processors, in turn, relay the message to T-1 others etc. In the last communication round, the message is sent to all (N-T-1) processors that did not receive the message before.

Since we can only use repetition codes here, the message size *msize* of all messages in the protocol is 1. In the following we will calculate the number of messages *mess* that is required for the case N ≥ 2T+1 respectively N < 2T+1.

The case N ≥ 2T+1.
Here, the required number of messages *mess* is:

$$mess = \left(\sum_{i=1}^{T} (T+1)^{i}\right) + (T+1)^{T} \cdot (N-T-1)$$

The first term expresses the number of messages in the first T communication rounds, whereas the second term expresses the number of messages sent in round T+1.

The case T+2 ≤ N < 2T+1.
The required number of messages *mess* is:

$$\left(\sum_{i=1}^{N-T-1} (T+1)^{i}\right) +$$
$$\left(\sum_{i=N-T}^{T+1} (T+1)^{N-T-1} \cdot (i-N+T+1)! \cdot \left\lceil \frac{T}{i-N+T+1} \right\rceil\right)$$

The first term expresses the number of messages in the first N-T-1 communication rounds, whereas the second term expresses the number of messages sent in the remaining rounds.

3.2.4.5. The Maximum Coding protocol

The number of messages that is exchanged between the processors during the multicast process can be calculated as follows. We will restrict ourselves to protocols in which the choice of the code only depends on the round in which the encoding of the messages is performed. Let a code used during round t (where $1 \le t \le T$) consist of code words of $n_c(t)$ symbols, obtained from data words of $k_c(t)$ symbols, let the symbol size be $b_c(t)$ bits. Then, the total amount of messages that is transmitted during the multicast process in terms of the number of unit messages, is (cf. [Krol91]):

$$mess = (N - T - 1) \cdot \prod_{t=1}^{T} \left(\frac{n_c(t)}{k_c(t)} \right) + \sum_{t=1}^{T} \prod_{i=1}^{t} \left(\frac{n_c(i)}{k_c(i)} \right)$$

Now, we will derive an expression for the minimal size *msize* of the message in the source. In every round t (where $2 \le t \le T$), the product of the message symbol size and the number of symbols must be equal to the size of the data word in round t-1. Thus,

$$(\forall\ t \in [2, T] :: b_c(t\text{-}1) = k_c(t) \cdot b_c(t)) \qquad (24)$$

For the original message in the source, we require:

$$msize = k_c(1) \cdot b_c(1) \qquad (25)$$

Moreover, the symbol size of each code must be sufficiently large, so in the case of MDS codes, we need to satisfy

$$\begin{aligned} &(\forall\ t \in [1, T] :: ((k_c(t) = 1 \Rightarrow b_c(t) \ge 1) \wedge \\ &(k_c(t) \ge 2 \Rightarrow b_c(t) \ge {}^2\log(n_c(t) - 1)))) \end{aligned} \qquad (26)$$

From relations (24), (25), and (26), *msize* can be calculated.

3.2.4.6. Results

In Table 3.2 through 3.5, we compare the amount of data communication needed by the various BAPs. Here, the Minimum Direction and Maximum Coding protocols are indicated by MinDir and MaxCod respectively. From these results, we conclude that for a small number of processors, the Dolev-protocol is to be preferred, but for larger number of processors the Maximum Coding protocol is favorable. When fault-tolerant services are implemented using the state machine approach [Schn90], the system should consist of at least $2T$+1 processors. For $N \ge 2T + 1$, if the system consists of enough processors, the Maximum Coding protocol is more efficient than the Dolev-protocol with regard to the relative number of messages required (See Table 3.6). E.g., for $T = 2$, the minimally required relative number of messages *mess* for the Maximum Coding protocol is 14.1 (for $N = 8$), whereas for the Dolev-protocol this number is 24 (for $N = 5$). However, the minimally required size *msize* of the initial message in the Maximum Coding protocol grows with an increasing number of processors N. For large *N* this initial message size may become impractical.

Table 3.2: *T* = 1

Protocol	*N*	*msize*	*mess*	applied codes [*n*,*k*,*b*]
Lamport	3	1	4	
Dolev	3	1	4	
MinDir	3	1	4	[2,1,1]
MaxCod	3	1	4	[2,1,1]
Lamport	4	1	9	
Dolev	4	1	9	
MinDir	4	1	6	[2,1,1]
MaxCod	4	2	4.5	[3,2,1]
Lamport	5	1	16	
Dolev	5	1	13	
MinDir	5	1	8	[2,1,1]
MaxCod	5	6	5.3	[4,3,2]
Lamport	16	1	256	
Dolev	16	1	57	
MinDir	16	1	30	[2,1,1]
MaxCod	16	56	16.1	[15,14,4]

Table 3.3: *T* = 2

Protocol	*N*	*msize*	*mess*	applied codes [*n*,*k*,*b*]
Lamport	4	1	15	
Dolev	4	1	12	
MinDir	4	1	15	[3,1,1]-[2,1,1]
MaxCod	4	1	15	[3,1,1]-[2,1,1]
Lamport	5	1	40	
Dolev	5	1	24	
MinDir	5	1	30	[3,1,1]-[3,1,1]
MaxCod	5	4	20	[4,2,2]-[3,1,1]
Lamport	6	1	85	
Dolev	6	1	40	
MinDir	6	1	39	[3,1,1]-[3,1,1]
MaxCod	6	12	15	[5,3,4]-[4,2,2]
Lamport	8	1	259	
Dolev	8	1	62	
MinDir	8	1	57	[3,1,1]-[3,1,1]
MaxCod	8	60	14.1	[7,5,12]-[6,4,3]
Lamport	16	1	2955	
Dolev	16	1	150	
MinDir	16	1	129	[3,1,1]-[3,1,1]
MaxCod	16	624	20	[15,13,48]-[14,12,4]

Table 3.4: $T = 3$

Protocol	*N*	*msize*	*mess*	applied codes [*n*,*k*,*b*]
Lamport	5	1	64	
Dolev	5	1	24	
MinDir	5	1	64	[4,1,1]-[3,1,1]-[2,1,1]
MaxCod	5	1	64	[4,1,1]-[3,1,1]-[2,1,1]
Lamport	7	1	516	
Dolev	7	1	60	
MinDir	7	1	276	[4,1,1]-[4,1,1]-[4,1,1]
MaxCod	7	12	94	[6,3,4]-[5,2,2]-[4,1,1]
Lamport	9	1	2080	
Dolev	9	1	99	
MinDir	9	1	404	[4,1,1]-[4,1,1]-[4,1,1]
MaxCod	9	180	39.1	[8,5,36]-[7,4,9]-[6,3,3]
Lamport	14	1	19045	
Dolev	14	1	174	
MinDir	14	1	724	[4,1,1]-[4,1,1]-[4,1,1]
MaxCod	14	2880	29.4	[13,10,288]-[12,9,32]-[11,8,4]
Lamport	16	1	35715	
Dolev	16	1	204	
MinDir	16	1	852	[4,1,1]-[4,1,1]-[4,1,1]
MaxCod	16	5280	29.8	[15,12,440]-[14,11,40]-[13,10,4]

Table 3.5: $T = 4$

Protocol	*N*	*msize*	*mess*	applied codes [*n*,*k*,*b*]
Dolev	6	1	40	
MaxCod	6	1	272	[5,1,1]-[4,1,1]-[3,1,1]-[2,1,1]
Dolev	9	1	112	
MaxCod	9	72	370.7	[8,4,18]-[7,3,6]-[6,2,3]-[5,1,1]
Dolev	21	1	353	
MaxCod	21	174720	50.1	[20,16,10920]-[19,15,728]-[18,14,52]-[17,13,4]
Dolev	22	1	372	
MaxCod	22	285600	50	[21,17,16800]-[20,16,1050]-[19,15,70]-[18,14,5]
Dolev	23	1	391	
MaxCod	23	367200	50.03	[22,18,20400]-[21,17,1200]-[20,16,75]-[19,15,5]

Table 3.6: Minimal relative number of messages for different values of T with $N \geq 2T + 1$

T	Protocol	*N*	*msize*	*mess*
1	Dolev	3	1	4
	MaxCod	3	1	4
2	Dolev	5	1	24
	MaxCod	8	60	14.1

Table 3.6: Minimal relative number of messages for different values of T with $N \geq 2T + 1$

T	Protocol	N	*msize*	*mess*
3	Dolev MaxCod	7 14	1 2880	60 29.4
4	Dolev MaxCod	9 22	1 285600	112 50

3.2.5. Conclusion

Section 3.2. describes and proves a new class of authenticated Byzantine Agreement Protocols based on encoding messages into symbols of an erasure-correcting codes on basis of a class of algorithms called the Authenticated Dispersed Joined Communication algorithms. We showed that for practical values of N and T the above-described BAPs require considerably less data communication than existing BAPs based on authenticated messages.

3.3. Authenticated self-synchronizing Byzantine Agreement protocols

In this section, we give a description of authenticated self-synchronizing Byzantine Agreement protocols, as introduced in Section 3.1.6. These BAPs guarantee Byzantine Agreement under less strict synchronicity assumptions than existing BAPs.

This section is structured as follows. In Section 3.3.1., we first describe the difference between our authenticated self-synchronizing BAPs and regular authenticated BAPs found in literature. In Section 3.3.2., an informal description of an authenticated self-synchronizing BAP is given. In Section 3.3.3., we describe the so-called unforgeability properties. In Section 3.3.4., the possible fault behaviour of faulty processors is described. In Section 3.3.5., some definitions are given which are used in the description of the authenticated self-synchronizing BAPs. Sections 3.3.6. and 3.3.7. are devoted to a description of our authenticated self-synchronizing BAPs and our optimized authenticated self-synchronizing BAPs, respectively. The assumptions we make for these BAPs are given in Section 3.3.8. In Sections 3.3.9. and 3.3.10. respectively, we prove that the assumptions stated in Section 3.3.8. and the unforgeability properties in Section 3.3.3. hold, our authenticated self-synchronizing BAPs, and optimized authenticated self-synchronizing BAPs guarantee Byzantine Agreement, i.e., satisfaction of the interactive consistency conditions IC1 and IC2 formulated in Section 3.1.2. Finally, in Section 3.3.11., the authenticated self-synchronizing BAPs and the optimized authenticated self-synchronizing BAPs are compared with regard to the required number of messages and the protocol execution time.

3.3.1. Difference between authenticated self-synchronizing Byzantine Agreement Protocols and regular authenticated Byzantine Agreement Protocols

In this section, the difference between an authenticated self-synchronizing Byzantine Agreement Protocol (BAP) and regular authenticated BAPs found in literature (e.g., in [LaSP82, BaMD93, GoLR95, DoSt83, PoKr95, DwLS88]) will be given. In fact, an authenticated self-synchronizing BAP is nothing more than a regular authenticated BAP, which is made suitable for arbitrary ***synchronous*** systems. As is shown below, regular authenticated BAPs are usually only applicable in a special class of synchronous systems, called ***lock-step synchronous systems***.

Recall from Section 3.1.2., that we have defined a ***BAP*** as a protocol which is executed in a system consisting of a set, N, of N processors, up to T of which may behave maliciously. One of the processors, called the source, holds an initial value, which it communicates to the other processors in the system in a number of communication phases. In the presence of up to T faulty processors, the BAP guarantees Byzantine Agreement, i.e., satisfaction of the so-called ***interactive consistency conditions***:

IC1. All correct processors agree on the initial value they think they have received from the source.

IC2. If the source is correct, then the above-mentioned agreement equals the initial value actually sent by the source.

The interactive consistency conditions IC1 and IC2 are sometimes referred to as the ***agreement condition*** and the ***validity condition***, respectively.

The BAPs that we have considered consist of a multicast process and a decision-making process. In the considered BAPs, the multicast process consists of T+1 communication phases, in which the processors in the system exchange messages in order to reach agreement on the initial value held by the source. In the decision-making process, the correct processors decide on basis of the set of message values of the messages they have accepted during the multicast process.

An ***authenticated BAP*** is a BAP, in which, during the multicast process, correct processors sign every message they generate or relay with an unforgeable signature.

Recall that, in Section 3.1.4.1., we have defined a ***synchronous system*** as a system with the following two properties:

1. The rate at which the processor clock of any correct processor drifts from real-time is bounded by a factor (1+ρ). Such clocks are called ρ-bounded clocks. Processor clocks of faulty processors may run at arbitrary rates.
2. Furthermore, there exists a real-time upper bound (τ_{max}) on the time needed to communicate a message from a correct processor to another processor in the system.

We assume that there also exists a real-time lower bound (τ_{min}, with $\tau_{min} \leq \tau_{max}$) on the time needed to communicate a message from a correct processor to another processor in the system. We assume that ρ, τ_{max}, and τ_{min} are a priori known by each correct processor.

An ***authenticated self-synchronizing BAP*** is defined as a BAP which guarantees Byzantine Agreement (i.e., satisfaction of the interactive consistency conditions) in an arbitrary ***synchronous*** system (i.e., a synchronous system with arbitrary nonnegative ρ and τ_{max}, and arbitrary nonnegative $\tau_{min} \leq \tau_{max}$, and ρ, τ_{max} and τ_{min} a priori known by all correct processors), while allowing arbitrary clock skew between the processor clocks of the processors in the system, and without requiring the correct processors to know the start of the BAP and to start the BAP simultaneously.

In contrast, most authenticated BAPs found in literature require the system in which the BAP is executed to be a ***lock-step synchronous system*** (i.e., a synchronous system, in which $\rho = 0$, and $\tau_{min} = \tau_{max}$) and they require the correct processors in the system to know the start of the BAP and to start the BAP simultaneously. The designs of many authenticated BAPs found in literature implicitly make use of properties of such a lock-step synchronous system. Exploiting these properties simplifies the design of these BAPs considerably, but confines their applicability to lock-step synchronous systems, in which the start of the BAP is known by all correct processors.

Most authenticated BAPs that are designed for lock-step synchronous systems implicitly make use of the following two properties:

1. In a lock-step synchronous system, ***protocol synchronization*** between the correct processors in the system is easily achieved. By having all correct processors select their corresponding communication phases of the BAP to have equal lengths, it is guaranteed that every communication phase starts and ends simultaneously in all correct processors, since all correct processors start the BAP simultaneously.
2. Furthermore, in a lock-step synchronous system, the BAP can be designed in such a way that all valid messages arrive in a processor in a fixed a priori known order. Then, the ***path*** (i.e., the sequence of processors) along which a particular message *m* travelled from the source to a particular destination processor *c*, can be deduced from the time slot in which message *m* arrives in *c*. This information is needed in the multicast process, in order to enable a correct processor to determine for any received valid message *m* a set of processors which includes all correct processors that did not yet receive *m*, and to relay *m* to (some or all of) these processors.

Usually, in authenticated BAPs found in literature, it is automatically assumed that the BAP is executed simultaneously on all correct processors, and that a correct processor knows the path of every valid message it receives. Although the BAPs rely on these assumptions, no provisions have been made in the BAP to satisfy them.

In an arbitrary synchronous system (i.e., a synchronous system with arbitrary nonnegative ρ and τ_{max}, and arbitrary nonnegative $\tau_{min} \le \tau_{max}$), it is not allowed to rely on the two properties mentioned above to hold. In any BAP designed for such a system, protocol synchronization as well as knowledge about the path along which received valid messages have travelled should explicitly be achieved. In Section 3.1.1.1. and 3.1.1.2. respectively, it will be explained how these goals can be achieved.

3.3.1.1. Achieving protocol synchronization between correct processors in a BAP

A BAP runs on a system consisting of a set *N* of processors interconnected by a network. One of the processors is the source. The source wants to communicate a value to the other processors (the destination processors). In general, the source starts the protocol by multicasting a message to (some or all of) the destination processors in the network. At receipt of a valid message[14] (directly or indirectly) from the source, a correct

14. See Section 3.3.3.1. for a definition of a valid message.

destination processor will start its own sub-protocol. In order to prevent confusion, we will refer to the protocol as a whole as the ***BAP***, and to any ‘sub-protocol’ of the BAP running on a certain destination processor *p* as the ***sub-BAP of processor p***.

The sub-BAP of a *correct destination processor* starts at receipt of the first valid message that is received (directly or indirectly) from the source. The sub-BAP of the *source* starts as soon as the source decides to multicast its initial value to (some or all of) the destination processors in the network. At the start of its sub-BAP, a correct processor calculates the real-time instances at which each communication phase of its sub-BAP starts and ends, on basis of its own processor clock, and the globally known parameters ρ, τ_{min}, and τ_{max}. Notice that, because each processor has its own processor clock, and because each processor may start its sub-BAP at a different real-time instance, the real-time instances at which corresponding communication phases of sub-BAPs of different processors begin and end, may be different.

In general, the destination processors in the system do not know at which moment the source starts a BAP (since it may be responding to some external request). However, as soon as a destination processor receives in time[15] a valid message from a BAP (either from the source or from another destination processor), implicitly, this destination processor is informed that this BAP was started.

Any correct destination processor that receives in time a valid message from a BAP will start its sub-BAP and decide on basis of the message values of the valid messages it has accepted. If, however, any correct destination processor is not informed of the start of the BAP (because it does not receive a valid message), it will not start a sub-BAP and will not decide. So, ***as soon as any correct destination processor is informed of the start of the BAP, all other correct destination processors must also be informed of it, otherwise interactive consistency condition IC1 (the agreement condition) is violated***.

If the source is *correct*, it may broadcast the value it wanted to communicate to all destination processors[16]. Implicitly, all correct destination processors are informed that the protocol has started.

However, if the source is *faulty*, it may send a valid message to some destination processors while it does not send a valid message to others[17]. Let I be the set of destination processors that received a valid message from the source s, and $N - I - \{s\}$ be the set of destination processors that did not receive a valid message from s. Then, the processors in I are informed that the protocol has started, while the ones in $N - I - \{s\}$ aren’t. The processors in $N - I - \{s\}$ should still be informed of the start of the protocol. The

15. See Section 3.3.4. for a definition of ‘received in time’
16. For simplicity, we assume here, that a correct source broadcasts a valid message to all destination processors in the system (just as it is assumed in the BAPs described in Section 3.3.6.). In Section 3.3.7., we make clear that it is sufficient that each destination processor receives a valid message (directly or indirectly) within a finite amount of time after the source started the BAP.
17. If the source is faulty, it is not guaranteed that any correct destination processor ever receives a message from the BAP. At the end of this section we will show that this situation never leads to violation of the interactive consistency conditions.

processors in *I* may implicitly inform the correct processors in N - I - $\{s\}$ that the protocol has started by relaying the valid message they (i.e., the processors in *I*) received to all correct destination processors that did not yet receive it.

However, it is possible that *I* contains faulty processors only. In this case, the processors in *I* may relay their valid message to some processors in $N - I - \{s\}$, but not to all. Then, the processors in $N - I - \{s\}$ that received a valid message, should inform those destination processors that did not yet receive a valid message, etcetera.

From the foregoing, we see that *protocol synchronization* between all correct destination processors can be achieved as follows[18]. We assume that every correct destination processor starts the first communication phase of its sub-BAP at receipt of the first valid message from the BAP (see Section 3.3.6. and 3.3.7. for details) . This may be a message from the source, or a message that was relayed by another destination processor. In order to synchronize all destination processors, any correct destination processor *c* should relay the first valid message *m* from the protocol it has received (directly or indirectly) to the set of all correct destination processors that did not yet receive it[19]. Then, if any correct destination processor is ever informed of the start of a BAP, then all correct destination processors are informed of it within a bounded time interval. The details of the protocol and the proofs are given in the sequel.

One could argue that, if the source is faulty, it is not guaranteed that a correct destination processor ever receives a valid message of the BAP (e.g., because the source does not send any valid message to a correct destination processor) If a correct destination processor *c* never receives a valid message of a certain BAP, processor *c* does not start its sub-BAP and will not decide. In this case, no correct destination processor will decide, since if any correct destination processor would ever have started its sub-BAP, *c* would also have started its sub-BAP[20]. If the source is faulty and no correct destination processor ever decides, the interactive consistency conditions IC1 and IC2 are trivially satisfied. So, the only interesting case is that in which, during the BAP, there is at least one correct destination processor that receives a valid message.

An example of how protocol synchronization between correct processors is achieved, is illustrated in Figure 3.7. The example shows a faulty source, *s*, and three correct processors p_i, p_j, and p_k (with s, p_i, p_j, and $p_k \in N$). Since the source is faulty, it may send arbitrary messages at arbitrary times. The correct processors will not start a BAP, unless at least one of them receives at least one valid message. In the example, it is assumed that the source generates two valid messages, $m(mv,(s;p_i))$ and $m(mv',(s;p_j))$, which are sent to p_i and p_j, respectively. At receipt of the first valid message from the BAP, the correct processors start their sub-BAP. Thus, processor p_i starts its sub-BAP at receipt of message $m(mv,(s;p_i))$ from the source, and processor p_j (and p_k respec-

18. The underlying principle applied here is that of so-called ***diffusion induction***. This principle is formally described in [CASD95].
19. In Section 3.3.2., we show that by including so-called ***path information*** (see Section 3.3.2.) in each valid message *m*, a set of processors can be found, which includes all correct processors that did not yet receive message *m*.
20. This is implied by Lemma 3.6.1. respectively Lemma 3.9.1. in Sections 3.3.9. and 3.3.10. respectively.

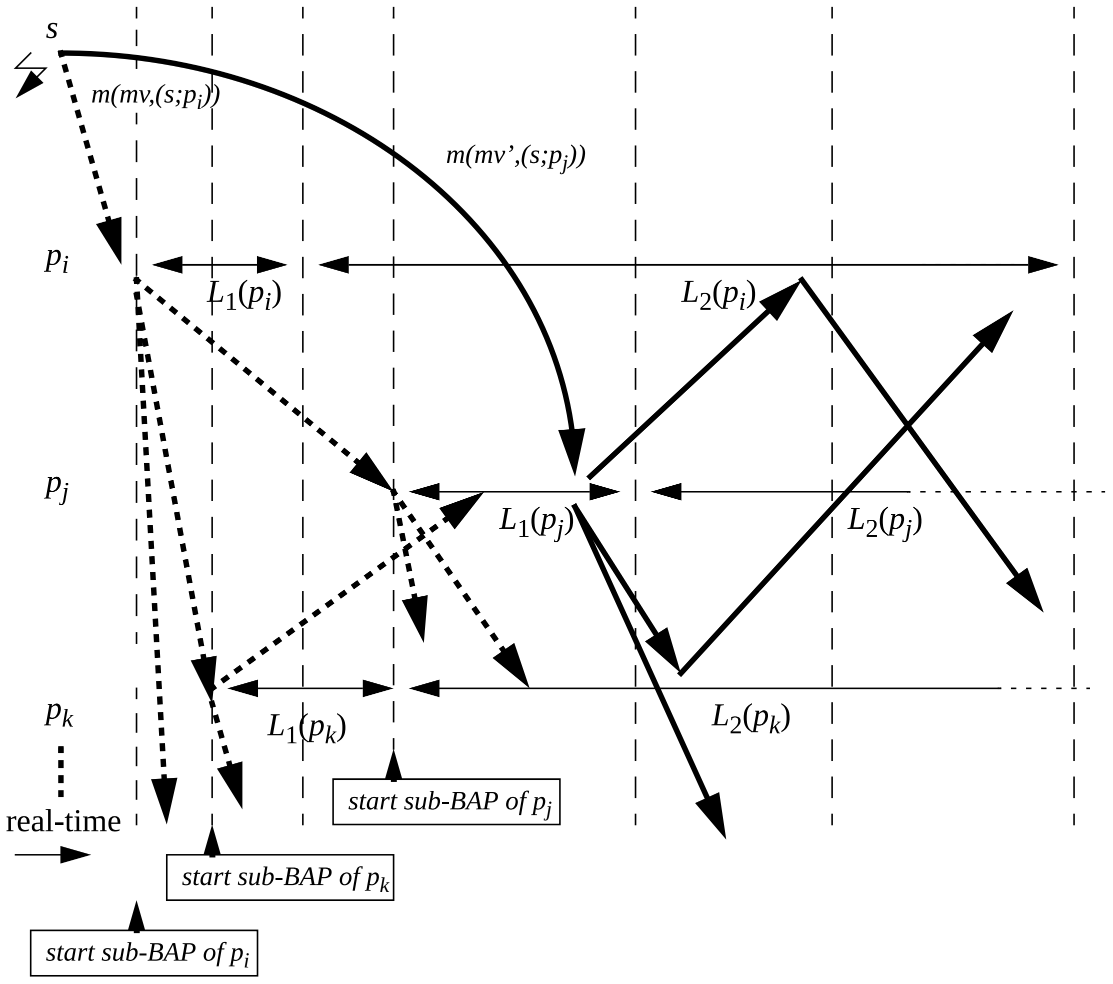


Figure 3.7. *Protocol synchronization between correct processors in an authenticated self-synchronizing BAP with a faulty source s by means of diffusion induction.*

tively) start their sub-BAP at receipt of message $m(mv,(s;p_i;p_j))$ (and $m(mv,(s;p_i;p_k))$, respectively) from p_i. Since processor p_i relays the message it has received from the source to all other processors, it is guaranteed that all correct processors will start their sub-BAP within a bounded time interval.

3.3.1.2. Path information

Lamport et al. were the first to formulate the problem of Byzantine Agreement [PeSL80] and to define some BAPs for synchronous systems to solve this problem [LaSP82]. Since then, a lot of more efficient solutions to this problem have appeared, many of them, however, being based on the BAPs formulated in [LaSP82].

The BAPs in [LaSP82] are based on the assumption that for every valid message *m* received by a correct processor *c*, *c* knows the ***path*** (i.e., the sequence of processors) along which *m* travelled from the source to *c*. This so-called ***path knowledge assumption*** is generally used in other BAPs, too (e.g., in [BaMD93, GoLR95, DoSt83, PoKr95, DwLS88]). It is needed in the multicast process, to enable a correct processor to determine for any received valid message *m* a set of processors which includes all correct processors that did not yet receive *m*, and to relay *m* to (some or all of) these processors. In some BAPs (e.g., in [PoKr95]), the path knowledge assumption is also needed in the decision-making process, in order to enable any correct processor *c* to group the messages it has received during the multicast process according to the path along which they travelled from the source to processor *c*.

In a lock-step synchronous system (i.e., a synchronous system in which $\rho = 0$, and $\tau_{min} = \tau_{max}$), a BAP may be designed in such a way, that all valid messages arrive in a processor in a fixed a priori known order. Then, the path along which a particular message *m* has travelled can be deduced from the time slot in which message *m* arrives in a particular destination processor.

In arbitrary synchronous systems (with arbitrary nonnegative ρ and τ_{max}, and arbitrary nonnegative $\tau_{min} \leq \tau_{max}$), message arrival times may be uncertain. As a consequence, the order in which message arrive in a particular destination processor may not be a priori known. Therefore, the path knowledge assumption does not automatically hold. In order to enable correct processors to obtain the required information, in the BAPs described in this section, so-called ***path information*** is included in each valid message. If this path information is ***correct,*** it describes the path along which the message travelled from the source to the current processor. However, there is no easy way to check if path information included in a valid message is indeed correct. It is only possible to check if path information satisfies several properties[21]. Path information that satisfies all these properties is called ***valid path information***.

Of course, malicious processors may try to undetectably corrupt the path information of messages they relay. By having every correct processor sign the path information of each message with an ***unforgeable signature***, the effects of corruption of this ***authenticated*** path information by malicious processors can be drastically restricted. By having all correct processors apply unforgeable signatures, it is more difficult (though not impossible) for faulty processors to transform correct path information into valid but incorrect path information.

We assume that a correct processor's signature cannot be forged. No assumptions are made about a faulty processor's signature. In particular, we allow a faulty processor's signature to be forged by another faulty processor, thereby permitting ***collusion*** among the faulty processors.

Provided that signatures of correct processors cannot be undetectably forged by malicious processors, the authenticated path information of a valid message *m only* contains the signature of a correct processor *c*, if processor *c did actually receive and accept* message *m* (or, if *c* is the source, that *c* did actually generate and multicast *m*). (This is expressed by the so-called ***unforgeability properties***, which will be proved to hold in Section 3.3.3.5.) Then, the set of processors, the signature of which is absent in the authenticated path information of *m*, includes all correct processors that did not yet receive message *m*. Thus, provided that signatures of correct processors cannot be

21. These properties are listed in the next section

undetectably forged by malicious processors, it is possible to deduce from the authenticated path information of a valid message *m* a set of processors, which includes all correct processors that did not yet receive *m*. By relaying *m* to this set of processors, it is guaranteed that *m* is sent to all correct processors that did not yet receive *m*.

For our authenticated self-synchronizing BAPs (described in Section 3.3.6 and 3.3.7. respectively), we assume[22] that every valid message *m* consists of two parts: the ***message value***, i.e., the information that the sender of *m* wanted to communicate, and authenticated path information of *m*. By integration of a message digest [Mull93] of the message value of a message *m* in the authenticated path information of *m*, malicious processors are prevented from undetectably combining the message value of one message with the path information of a different message.

Any valid message can be uniquely constructed given its message value and the path (i.e., the sequence of processor identifiers) described by the authenticated path information of the message, indicating the sequence of processors, of which the cryptographic functions have subsequently been used to sign the message, followed by the identification of the receiver of this message. Therefore, a valid message with message value *mv*, and path information describing a path ($\underline{\boldsymbol{p}}$) can be denoted by *m*(*mv*, ($\underline{\boldsymbol{p}}$)).

For path ($\underline{\boldsymbol{p}}$) in a valid message *m*(*mv*,($\underline{\boldsymbol{p}}$)), it holds that all processor identifiers in the path are different (i.e., a valid message is never signed twice by the same processor), and the length of the path is less than or equal to T+2 (i.e., a valid message does never carry more than T+1 signatures).

3.3.2. Informal description of an authenticated self-synchronizing BAP

In this section, we will give an informal description of an authenticated self-synchronizing BAP. This section is meant to give a general idea of how the protocol works. The details can be found in the formal description, which follows in Sections 3.3.3. through 3.3.6. The assumptions made for this protocol can be found in Section 3.3.8 whereas the proofs are denoted in Section 3.3.9.

We first consider the execution of the authenticated self-synchronizing BAP, in the case that all processors in the system behave correctly in Section 3.3.2.1. After that, in Section 3.3.2.2., we consider the execution of the authenticated self-synchronizing BAP in the presence of up to T maliciously behaving processors.

3.3.2.1. Execution of an authenticated self-synchronizing BAP in a system of correctly functioning processors only

In this case, a correctly functioning source starts the BAP by multicasting a valid message to all other processors in the system. At the same moment, the source starts its sub-BAP and calculates the real-time instances at which the various communication phases of its sub-BAP start and end, on basis of its own processor clock and the globally known parameters ρ, τ_{min}, and τ_{max}.

22. Assumption A3.6 in Section 3.3.8.

After a real-time period of at most τ_{max} after the source has started the BAP (by multicasting a message to all other processors in the system), every correct processor will have received a message from the source. As soon as a correct processor *c* receives a message from the source, processor *c* will check if the message satisfies the following conditions[23]:

1. First, processor *c* checks whether the received message is ***valid***, i.e., *c* checks whether the message is syntactically correct. Since the source is correct, it will only generate valid messages. The received message can be denoted as *m*(*mv*,($\underline{\boldsymbol{p}}$)), for some message value *mv*, and some path ($\underline{\boldsymbol{p}}$).
2. Then, processor *c* checks if its own identification is the last identification in path ($\underline{\boldsymbol{p}}$). Since the source is correct, this will always be the case.
3. Furthermore, processor *c* checks if it did not accept a valid message with path information describing the same path ($\underline{\boldsymbol{p}}$) before. Since the source is correct, this can never be the case.
4. Finally, processor *c* checks whether the received message has been ***timely received***. The BAP is designed in such a way that any message from a correct source sent to some correct processor *c*, is always timely received in *c*.

If the received message satisfies all these conditions, it is accepted by processor *c*. If the received message, say *m*, is the first valid message that *c* receives[24], then *c* starts its sub-BAP at receipt of *m*. In that case, processor *c* calculates the real-time instances at which the various communication phases of its sub-BAP start and end, on basis of its own processor clock, the moment at which the first valid message arrives, and the globally known parameters ρ, τ_{max}, and τ_{min}.

If processor *c* has accepted message *m*, then *c* ***relays*** *m* to all processors, the identification of which is not present in the path described by the path information of *m*. Processor *c* ***relays*** message *m* to another processor *d* by generating a new message *m*‘ from *m* (in a specific way described below), and by sending the new message *m*‘ to *d*. Message *m*‘ is generated from *m* as follows. First, the identification of *d* is concatenated to the

23. For the BAP to satisfy the interactive consistency conditions IC1 and IC2, it is not required that received messages satisfy the third condition, i.e., it is allowed to accept a valid message *m* with path information describing a path ($\underline{\boldsymbol{p}}$), while a different valid message *m*’ with path information describing a path ($\underline{\boldsymbol{p}}$) has already been accepted previously.

    Acceptance of message *m* is ***allowed*** but ***not required***, since, for messages *m* and *m*’ to be different, their message values must be different, which implies that the source is faulty. Hence, interactive consistency condition IC2 (the validity condition) is automatically satisfied, and we only need to guarantee satisfaction of interactive consistency condition IC1 (the agreement condition). If message *m* (with some message value *mv*) is *accepted* in the receiving correct processor, a valid message with message value *mv* will be accepted in all correct processors, if message *m* is *rejected*, it will not cause acceptance of a valid message with message value *mv* in any other correct processor. Hence, satisfaction of IC1 does not depend on the acceptance of message *m*.

    In the BAPs described in this chapter, we have chosen to reject any such message *m*, since this makes the BAPs more efficient with regard to the required communication overhead.

24. Notice that it is possible that processor *c* has already received a valid message (indirectly from the source, via an intermediate processor *d*) before it receives a valid message from the source *s*. This is possible, provided that communicating a message from *s* to *d*, and, subsequently, relaying this message from *d* to *c*, is faster than communicating a message from *s* to *c*.

path information of *m*. The result is signed with *c*'s own secret cryptographic function, yielding new path information. The concatenation of the message value of *m* and the new path information forms the message *m*' that is sent to *d*.

For any message *m* which a correct processor *c* receives during execution of its sub-BAP, *c* will check if the received message satisfies all four above-mentioned conditions. A valid message *m*(*mv*,(***p***)) which *c* receives in communication phase *i* of its sub-BAP, will be regarded by *c* as timely received iff the number of processor identifiers in path (***p***) is greater than *i*.

Provided that message *m* satisfies all four conditions mentioned above, *c* accepts message *m* and relays *m* to all processors, the identification of which is not present in the path described by the path information of message *m*. The set of processors to which *m* is relayed is guaranteed to include all correct processors that did not yet receive *m*, since the authenticated path information of message *m* **only** contains the signature of a correct processor if this correct processor **did actually receive and accept** *m*.

After a correct processor has terminated the last communication phase of its sub-BAP, it decides on basis of the message values of the messages of the corresponding BAP that it has accepted during execution of its sub-BAP. If the message values of all accepted messages are equal, then one of the message values is taken as decision, if the accepted messages contain two or more different message values, then a default value is taken as decision.

3.3.2.2. Execution of an authenticated self-synchronizing BAP in the presence of up to *T* maliciously behaving processors

We will now consider the execution of an authenticated self-synchronizing BAP in the presence of up to *T* maliciously functioning processors. Collusion between faulty processors is allowed, i.e., they may use each other's signature to sign messages, and they may exchange messages. Furthermore, they may generate or relay messages at any time.

In the presence of faulty processors in the system, we will consider a BAP as being started as soon as at least one correct processor has started its sub-BAP. The start of the BAP may be initiated either by a correct processor, or by a set of colluding faulty processors. We distinguish between these two cases and show that in both cases, the interactive consistency conditions are satisfied.

We first consider the case that the BAP is initiated by a *correct* source *s*.

Case A: The BAP is initiated by a correct source *s*

In this case, the source *s* initiates the BAP by multicasting a valid message to all other processors in the system. Processor *s* starts its sub-BAP and accepts a copy of its own message in order to use it in the decision-making process. After a real-time period of at most τ_{max} after the source *s* has initiated the BAP, every correct processor *c* in the system will have received a valid message *m*(*mv*,(*s;c*)) from the source *s*. At receipt of this message, processor *c* will check if the message satisfies all four conditions mentioned in the previous section, which will always be the case since the source *s* is correct. Hence, any processor *c* will accept the message *m*(*mv*,(*s;c*)) received from the source.

If *m*(*mv*,(*s;c*)) is the first valid message that *c* receives, *c* will start its sub-BAP at receipt of *m*(*mv*,(*s;c*)).

As soon as a correct processor *c* has terminated the last communication phase of its sub-BAP, it will decide on basis of the set of message values of the messages it has accepted. Any correct processor *c* has at least accepted message *m*(*mv*,(*s;c*)) received from the source *s*. For any correct processor *c*, the set of message values on which *c* will base its decision will thus at least contain message value *mv* of the message received from the source. However, the set of message values on which *c* bases its decision cannot contain other message values *mv*‘ ≠ *mv*, because the source *s* is correct and hence, only generates and multicasts valid messages with message value *mv*, and faulty processors are not able to generate and multicast valid messages *m*(*mv*‘, (*s;c*)) with *mv*‘ ≠ *mv*, since such messages need the signature of processor *s*, which the faulty processors are unable to produce. Hence, any correct processor will decide *mv*, and thus, both interactive consistency conditions IC1 and IC2 are satisfied. ❏

We will now consider the case that the BAP is initiated by a *set of colluding faulty processors*.

Case B: The BAP is initiated by a set of colluding faulty processors
In this case, the source of the BAP is faulty, and thus, interactive consistency condition IC2 (the validity condition) is trivially satisfied. So, we only have to prove that interactive consistency condition IC1 (the agreement condition) is also satisfied.

As stated before, a BAP will only be considered as being started, if at least one correct processor has started its sub-BAP. If the BAP is not being started, then this implies that no correct processor ever starts its sub-BAP, and hence, no correct processor will ever decide. In this case, the interactive consistency conditions are trivially satisfied. It remains to consider the case in which at least one correct processor starts its sub-BAP.

We will show that, provided that at least one correct processor starts its sub-BAP and hence, will decide, all correct processors will start their sub-BAP and decide. Furthermore, we show that the decisions taken in all correct processors are equal (i.e., interactive consistency condition IC1 is satisfied).

In general, a correct processor will only start its sub-BAP either in case it acts as the source in a BAP, or as soon as it receives the first valid message of the BAP. In case the BAP is initiated by a set of colluding faulty processors, the source must be a faulty processor, thus, no correct processor can act as the source, and hence, a correct processor can only start its sub-BAP in reaction to the receipt of a valid message of the BAP. So, at least one correct processor must have received a valid message of the BAP.

Assume that processor *c* is the first correct processor that receives a valid message *m*(*mv*,(***q***;*c*)) from the BAP. Notice that path (***q***) can only contain identifications of faulty processors, since, if (***q***) would contain the identification of a correct processor *x* (say (***q***) = (***p***;*x*;***r***)), this correct processor *x* must have received and multicast a valid message *m*(*mv*,(***p***;*x*)) before *c* received *m*(*mv*,(***q***;*c*)), because faulty processors are not capable of generating a message signed with *x*’s signature.

At receipt of message $m(mv,(\boldsymbol{q};c))$, processor c will accept the message, start its sub-BAP, and relay the message to all processors, the identification of which is not present in path $(\boldsymbol{q};c)$. This implies that c relays $m(mv,(\boldsymbol{q};c))$ to all other correct processors in the system, since, as discussed above, path $(\boldsymbol{q};c)$ does not contain the identification of any *correct* processor except the identification of processor c. Hence, after a real-time period of at most τ_{max} after c has relayed message $m(mv,(\boldsymbol{q};c))$, it is guaranteed that *all other correct processors* in the system have also received the first valid message of the BAP.

The lengths of the communication phases of the sub-BAPs executed on the different processors in the system have been selected such that any valid message $m(mv,(\boldsymbol{q};c))$ accepted and relayed from any correct processor c to any other correct processor d during communication phase i $(1 \le i \le T)$ of c's sub-BAP will arrive in or before communication phase i+1 of d's sub-BAP in processor d. Message $m(mv,(\boldsymbol{q};c))$ arriving in communication phase i of c's sub-BAP is accepted by c iff the length of path $(\boldsymbol{q};c)$ is greater than i. Relaying $m(mv,(\boldsymbol{q};c))$ to processor d in communication phase i of c's sub-BAP will result in $m(mv,(\boldsymbol{q};c;d))$ being sent to d, and arriving in d in or before communication phase i+1 of d's sub-BAP. Since message $m(mv,(\boldsymbol{q};c))$ was accepted in processor c, we know that the length of path $(\boldsymbol{q};c)$ is greater than i. Hence, the length of path $(\boldsymbol{q};c;d)$ is greater than i+1, implying that message $m(mv,(\boldsymbol{q};c;d))$ is accepted in processor d, since $m(mv,(\boldsymbol{q};c;d))$ arrives in d in or before communication phase i+1 of d's sub-BAP.

Thus, message $m(mv;(\boldsymbol{q};c))$, which is accepted by c and relayed to any correct processor d will be accepted in any processor d, and, if this message is the first valid message that processor d receives, d will start its sub-BAP upon receipt of this message from c. Hence, at or before τ_{max} after processor c has started its sub-BAP, all other correct processors have also started their sub-BAP.

We will now show that interactive consistency condition IC1 is satisfied, i.e., all correct processors decide on basis of the same set of message values.

The decision taken in a correct processor is based on the set of message values of the messages that it has accepted during the various communication phases of its sub-BAP. The decisions taken in the correct processors will be the same, provided that the sets of message values obtained by all correct processors are the same.

These sets of message values are the same for all correct processors, provided that a message value present in the set of message values of any correct processor, is also present in the sets of message values of all other correct processors. A message value mv is present in the set of message values of a correct processor c, iff processor c has accepted a valid message $m(mv,(\boldsymbol{q};c))$, for some non-empty path $(\boldsymbol{q})$.

Assume that processor c has accepted some valid message $m(mv,(\boldsymbol{q};c))$. We will now show that it is guaranteed that any correct processor d will also accept some valid message $m(mv,(\boldsymbol{p};d))$, for some non-empty path $(\boldsymbol{p})$. This implies that, if some correct processor c includes mv in its set of message values on which its decision is based, then mv is also included in the set of message values of all other correct processors. We distinguish between the case that the length of path $(\boldsymbol{q};c)$ is smaller than T+2, and the case that the length of path $(\underline{\boldsymbol{q}};c)$ is equal to $T+2$.

Case 1: the length of path $(\underline{\boldsymbol{q}};c)$ is smaller than $T+2$.
Assume the length of path $(\underline{\boldsymbol{q}};c)$ is smaller than $T+2$. In this case, after having accepted message $m(mv,(\underline{\boldsymbol{q}};c))$, processor c relays this message to all processors, the identification of which is not present in path $(\underline{\boldsymbol{q}};c)$. Above, we have already seen that any correct processor x $(x \neq c)$, the identification of which is present in path $(\underline{\boldsymbol{q}};c)$ has already received and accepted a message $m(mv,(\underline{\boldsymbol{p}};x))$, with $(\underline{\boldsymbol{q}};c) = (\underline{\boldsymbol{p}};x;\underline{\boldsymbol{r}})$, before c received and accepted $m(mv,(\underline{\boldsymbol{q}};c))$. Any correct processor d, the identification of which is not present in path $(\underline{\boldsymbol{q}};c)$, will receive a message $m(mv,(\underline{\boldsymbol{q}};c;d))$ from c, which it accepts. So, if a correct processor c accepts a message with path information describing a path of length smaller than $T+2$ and message value mv, all correct processors will accept a message with message value mv.

Case 2: the length of path $(\underline{\boldsymbol{q}};c)$ is equal to $T+2$.
Assume the length of path $(\underline{\boldsymbol{q}};c)$ is equal to $T+2$. Since $m(mv,(\underline{\boldsymbol{q}};c))$ is a valid message, all processors, of which the identification is present in path $(\underline{\boldsymbol{q}};c)$, are different. Since it is assumed that the system contains at most T faulty processors, path $(\underline{\boldsymbol{q}};c)$ must contain the identification of at least 2 correct processors, hence, since processor c was assumed to be correct, path $(\underline{\boldsymbol{q}})$ must contain the identification of at least one correct processor. Since signatures of correct processors cannot be forged, this means that at least one correct processor x must have received and accepted a message $m(mv,(\underline{\boldsymbol{p}};x))$, with $(\underline{\boldsymbol{q}};c) = (\underline{\boldsymbol{p}};x;\underline{\boldsymbol{r}})$, and thus the length of path $(\underline{\boldsymbol{p}};x)$ is smaller than $T+2$. So, if a correct processor c accepts a message with path information describing a path of length $T+2$, there exists a processor x which has accepted a message with path information describing a path of length smaller than $T+2$ and message value mv. From case 1, it follows that all correct processors will accept a message with message value mv.

Thus, if any correct processor accepts a message with message value mv, all other correct processors will also accept a valid message with message value mv. Hence, the sets of message values of all correct processors, on which they base their decision, are always equal, and thus, all correct processors will always decide the same value. Hence, interactive consistency condition IC1 (the agreement condition) is always satisfied. This concludes our discussion, since we had already concluded that interactive consistency condition IC2 (the validity condition) is always satisfied. ❏

Thus, in both cases, both interactive consistency conditions IC1 and IC2 are satisfied.

3.3.3. The unforgeability properties

In authenticated BAPs found in literature, it is usually assumed that any alteration of the contents of messages signed by one or more correct processors can be detected (See Section 3.1.4.). In an arbitrary synchronous system (i.e., a synchronous system with arbitrary nonnegative ρ and τ_{max}, and arbitrary nonnegative $\tau_{min} \leq \tau_{max}$, and ρ, τ_{max}, and τ_{min} a priori known by all correct processors), it is impossible to detect all alterations of the contents of messages signed by one or more correct processors. However, as will be proved in Section 3.3., for our authenticated self-synchronizing BAPs and optimized authenticated self-synchronizing BAPs to guarantee Byzantine Agreement, it is sufficient to prove that the so-called unforgeability properties hold. The ***unforgeability properties*** UP1 and UP2 are defined as follows:

UP1 If a processor d receives a valid message $m(mv, (s; \underline{\mathbf{p}}; c; \underline{\mathbf{q}}; d))$, where c is the identification of a correct processor, then processor c has received and accepted a valid message $m(mv, (s; \underline{\mathbf{p}}; c))$.

UP2 If a processor d receives a valid message $m(mv, (s; \underline{\mathbf{q}}; d))$, where s is the identification of a correct processor, then processor s has generated and multicast a valid message $m(mv, (s; x))$ to the selected processors x in the system[25], and it has accepted a copy of the message itself.

As will be proved in Section 3.3.3.5., the unforgeability properties UP1 and UP2 hold, provided that several assumptions with regard to the authentication of messages, stated in Section 3.3.3.4., are satisfied.

However, before we are able to give these proofs, we first have to give a definition of valid messages, we have to explain how a correct source generates messages, and how a correct processor relays a message, and, finally, we have to define what we mean by unforgeable messages. Therefore, this section is structured as follows.

First, in Section 3.3.3.1., we give a definition of a ***valid message***[26]. Then, in Section 3.3.3.2. and 3.3.3.3. respectively, a description of the ***message generation process*** and the ***message relay process*** is given.

In Section 3.3.3.4., we present a number of assumptions with regard to the authentication of valid messages, and, in Section 3.3.3.5., we prove that under these authentication assumptions, the unforgeability properties UP1 and UP2 hold. Some final remarks are made in Section 3.3.3.6.

Before we start with Section 3.3.3.1., a final remark on the proof in Section 3.3.3.5. will be made.

The proof relies on the assumption that all messages signed by one or more correct processors are ***valid***. In order to make sure that all messages signed by one or more correct processors are valid, it must be guaranteed that *if a correct processor signs a message, the resulting signed message is a valid message.*

This requirement can be satisfied by ensuring that:

Rq1. a correct source only generates valid signed messages, and

Rq2. a correct processor only signs a message if this message is valid, and

Rq3. the result of signing a valid message by a correct processor is again a valid message.

25. In an authenticated self-synchronizing BAP (which will be described in Section 3.3.6.), a correct source multicasts a valid message to all other processors in the system. In an optimized authenticated self-synchronizing BAP (which will be described in Section 3.3.7.), a correct source multicasts a valid message to all processors in a subset selected by the subset selection rules of the BAP.

26. In general, a message is just a sequence of bits. Roughly speaking, a valid message can be seen as a sequence of bits in which the bits are ordered in such a way that meaningful information (such as the path information of the message and its message value) can be derived from it.

Requirement Rq1 can be satisfied by simply *defining* that a correct source only generates valid signed messages. The ***message generation process*** (as it is performed by a correct source), which is formally described in Section 3.3.3.2., ensures this. Requirement Rq2 and Rq3 respectively, can be satisfied by carefully defining the process of relaying a message by a correct processor. The description of this ***message relay process*** (as it is performed by a correct processor) can be found in Section 3.3.3.3.

3.3.3.1. Valid messages

Let N be the set of processors on which a BAP is executed. In general, during the multicast process of a BAP, every processor $a \in N$ receives a number of messages, say M_a ($M_a \geq 0$). Let *Messages* be the (infinite) set of all possible messages. For any processor $a \in N$, and $1 \leq i \leq M_a$, by $mess_i(a)$, we will denote the i-th message that processor a receives during the multicast process of the BAP. Here, $mess_i(a) \in Messages$.

Such a message $mess_i(a)$ needs not always be valid. As we will see later, faulty processors may corrupt a valid message, which may result in an invalid message. In this section, we will formally define what we mean by valid respectively invalid messages:

D3.1. Any message $mess_i(a) \in Messages$ (with $1 \leq i \leq M_a$) received by a certain processor $a \in N$ is a ***valid message*** iff $mess_i(a) \in ValidMessages$. Conversely, any message that is not valid is defined as an ***invalid message***.

Here, *ValidMessages* is the set of all possible valid messages, which will be defined below.

As already stated in Section 3.3.1.2., we assume that every valid message m consists of two parts: the message value, i.e., the information that the sender of m wanted to communicate, and authenticated path information of m. Furthermore, a message digest of the message value of a valid message m is integrated in the path information of m.

If the path information of a received valid message m is correct, it describes the path along which message m travelled from the source to the current processor that has received the message. All processors are included in the path information, starting from the source and ending with the receiver of the message.

A path which contains no processors is called the ***empty path***.

Recall from Section 3.2.1.1., that for paths, we use the following notation:

N3.1.	( )	denotes the empty path
	(a)	denotes a path consisting of processor a
	($\underline{\mathbf{p}}$;a)	denotes a path consisting of path ($\underline{\mathbf{p}}$) concatenated with processor a.

The path ($\underline{\mathbf{p}}$;a) describes the path that follows all processors in path ($\underline{\mathbf{p}}$) (in order from left to right) and ends in processor a.

In order to be able to define the set *ValidMessages*, we first have to define several other sets. It is assumed that all sets defined in this section are disjoint.

In this section, the following sets will subsequently be defined:

• *MessageValues*	the set of all possible message values	(see D3.2)
• *Paths*	the set of all possible paths	(see D3.3)
• *ValidPaths*	the set of all possible valid paths	(see D3.8)
• *MessageDigests*	the set of all possible message digests	(see D3.9)
• *EncryptedPathParts*	the set of all possible encrypted path parts	(see D3.11)
• *ValidPathParts*	the set of all possible valid path parts	(see D3.12)
• *ValidMessages*	the set of all possible valid messages	(see D3.15)

As will be shown below, a valid message is a composition of elements of the above-mentioned sets. In order to make it easier to understand the meaning of the above-mentioned sets, below, we will present an example of a valid message, in which all elements are clearly specified.

Example:
In this example, we consider the composition of a valid message $m(mv, (s;a;b))$, i.e., a valid message with message value mv and path information describing a path $(s;a;b)$. (In a system of correctly functioning processors only, such a message is the result of generation of a valid message $m(mv,(s;a))$ by source s, which is sent to processor a, and relayed from processor a to processor b.)

Message $m(mv, (s;a;b))$ has the form:

$$(mv; ((s;a); K_{(a)}\{K_{(s)}\{digest(mv); a\}; b\}))$$

The message consists of two concatenated parts: a message value $mv \in$ *MessageValues*, and a valid path part $((s;a); K_{(a)}\{K_{(s)}\{digest(mv); a\}; b\}) \in$ *ValidPathParts*. The valid path part, in its turn, consists of two concatenated parts: a valid path $(s;a) \in$ *ValidPaths*, denoting the sequence of processors that have subsequently signed the message[27], and an encrypted path part $K_{(a)}\{K_{(s)}\{digest(mv); a\}; b\} \in$ *EncryptedPathParts*. This encrypted path part consists of the concatenation (encrypted with the secret cryptographic function $K_{(a)}$ of processor $a \in N$) of another encrypted path part $K_{(s)}\{digest(mv); a\} \in$ *EncryptedPathParts* and the identification of a processor $b \in N$. The encrypted path part $K_{(s)}\{digest(mv); a\}$, in its turn, consists of the concatenation (encrypted with the secret cryptographic function $K_{(s)}$ of processor $s \in N$) of a message digest $digest(mv) \in$ *MessageDigests* of message value $mv \in$ *MessageValues*, and the identification of processor $a \in N$. Function *digest* is a function on message values, which yields the message digest of this message value. Notice that, in this way a message digest of the message value of $m(mv, (s;a;b))$ is integrated in the path information of $m(mv, (s;a;b))$.

Before we start with the definitions of the sets, we briefly investigate how messages are generated respectively relayed.

As stated before, message $m(mv, (s;a;b))$ is constructed by processor $a \in N$, when ***relaying*** message $m(mv, (s;a))$ to b. Message $m(mv, (s;a))$ has been ***generated*** by the

27. Although this sequence of processors is not strictly necessary in a valid message, it is added for convenience, since it greatly simplifies the effort required to decrypt the encrypted path part, for the sequence of processors that have subsequently signed the message, is known.

source s and has the form:

$$(mv, ((s); K_{(s)}\{digest(mv); a\}))$$

Hence, message $m(mv, (s;a))$ consists of two parts: a message value $mv \in MessageValues$, and a valid path part $((s); K_{(s)}\{digest(mv); a\}) \in ValidPathParts$, which consists of a concatenation of a valid path $(s) \in ValidPaths$ and an encrypted path part $K_{(s)}\{digest(mv); a\} \in EncryptedPathParts$. It is easy to observe that the different components of message $m(mv,(s;a))$ are also part of message $m(mv, (s;a;b))$. Relaying message $m(mv, (s;a))$ to b is done by constructing $m(mv,(s;a;b))$ from $m(mv,(s;a))$ as follows.

The message value of $m(mv, (s;a;b))$ is identical to that of $m(mv, (s;a))$. The valid path part consists of a concatenation of a path of encryptors $(s;a) \in ValidPaths$, and an encrypted path part $K_{(a)}\{K_{(s)}\{digest(mv); a\}; b\}$. The path of encryptors $(s;a)$ is constructed by concatenating the path of encryptors (s) of $m(mv, (s;a))$ with the identification of processor a. The encrypted path part $K_{(a)}\{K_{(s)}\{digest(mv); a\}; b\}$ is constructed by concatenating the encrypted path part $K_{(s)}\{digest(mv); a\}$ of $m(mv, (s;a))$ with the identification of processor b, and encrypting the concatenation with the secret cryptographic function $K_{(a)}$. ❑

Now, we are able to start with the definitions:

D3.2. It is difficult to formally define *MessageValues* (the set of all possible message values). In general, a message value will be a sequence of bits, sometimes with a predefined bit pattern. The set of possible message values depends on the specific implementation that is considered. Therefore, we simply define:

$$MessageValues = \{\, mv \mid mv \text{ is a possible message value}\}$$

D3.3. The (infinite) set *Paths* is defined as follows:

$$\begin{aligned}&(\forall\, (\underline{\boldsymbol{p}}) :: (\underline{\boldsymbol{p}}) \in Paths \equiv\\ &\qquad ((\underline{\boldsymbol{p}}) = (\,)) \vee\\ &\qquad (\exists\, (\underline{\boldsymbol{q}}) \in Paths : \exists\, a \in N :: (\underline{\boldsymbol{p}}) = (\underline{\boldsymbol{q}}\,; a)))\end{aligned}$$

The following functions are defined on paths:

D3.4. $(\forall\, (\underline{\boldsymbol{p}}) \in Paths :: empty((\underline{\boldsymbol{p}})) \equiv (\underline{\boldsymbol{p}}) = (\,))$

i.e., function $empty((\underline{\boldsymbol{p}})) \equiv$ TRUE iff path $(\underline{\boldsymbol{p}})$ is the empty path.

D3.5.

$$\begin{aligned}&((\forall\, (\underline{\boldsymbol{p}}) \in Paths :: empty((\underline{\boldsymbol{p}})) \Rightarrow length((\underline{\boldsymbol{p}})) = 0) \wedge\\ &\;(\forall\, (\underline{\boldsymbol{p}}) \in Paths : \forall\, (\underline{\boldsymbol{q}}) \in Paths : \forall\, a \in N ::\\ &\qquad ((\underline{\boldsymbol{p}}) = (a; \underline{\boldsymbol{q}}) \vee (\underline{\boldsymbol{p}}) = (\underline{\boldsymbol{q}}; a)) \Rightarrow length((\underline{\boldsymbol{p}})) = 1 + length((\underline{\boldsymbol{q}})))\\ &)\end{aligned}$$

i.e., function $length((\underline{\boldsymbol{p}}))$ returns the length of path $(\underline{\boldsymbol{p}})$, i.e., the number of processors in path $(\underline{\boldsymbol{p}})$.

D3.6.

$$\begin{aligned}&((\forall\, (\underline{\boldsymbol{p}}) \in Paths : \forall\, a \in N :: empty((\underline{\boldsymbol{p}})) \Rightarrow occurrences(a, (\underline{\boldsymbol{p}})) = 0) \wedge\\ &\;(\forall\, (\underline{\boldsymbol{p}}) \in Paths : \forall\, (\underline{\boldsymbol{q}}) \in Paths : \forall\, a \in N ::\\ &\qquad ((\underline{\boldsymbol{p}}) = (\underline{\boldsymbol{q}}; a)) \Rightarrow occurrences(a, (\underline{\boldsymbol{p}})) = 1 + occurrences(a, (\underline{\boldsymbol{q}}))) \wedge\\ &\;(\forall\, (\underline{\boldsymbol{p}}) \in Paths : \forall\, (\underline{\boldsymbol{q}}) \in Paths : \forall\, a,b \in N ::\end{aligned}$$

$$(((\underline{\boldsymbol{p}}) = (\underline{\boldsymbol{q}}; b)) \wedge (a \neq b)) \Rightarrow$$
$$occurrences(a, (\underline{\boldsymbol{p}})) = occurrences(a, (\underline{\boldsymbol{q}})))$$

)

i.e., function *occurrences*(a, ($\underline{\boldsymbol{p}}$)) returns the number of times that processor a occurs in path ($\underline{\boldsymbol{p}}$).

D3.7. $(\forall\ (\underline{\boldsymbol{p}}) \in Paths :: validpath(\ (\underline{\boldsymbol{p}})\ ) \equiv$
$(length(\ (\underline{\boldsymbol{p}})\ ) \le T{+}2 \wedge (\forall\ a \in N :: occurrences(a, (\underline{\boldsymbol{p}})) \le 1)))$
i.e., function *validpath*(($\underline{\boldsymbol{p}}$)) ≡ TRUE if the length of path ($\underline{\boldsymbol{p}}$) is less than or equal to T+2, and every processor a occurs at most once in path ($\underline{\boldsymbol{p}}$).

Now, the set of all possible valid paths *ValidPaths* is defined by:

D3.8. $ValidPaths = \{\ (\underline{\boldsymbol{p}}) \in Paths \mid validpath(\ (\underline{\boldsymbol{p}})\ )\ \}$

Furthermore, the set of all possible message digests *MessageDigests* is defined by:

D3.9. $MessageDigests = \{md \mid md = digest(mv) \wedge mv \in MessageValues\ \}$

Here, *digest* is a message digest function known by all correct processors. Application of this function on a message value results in a message digest of that message value. Assumption As2 (in Section 3.3.3.4) expresses the required properties of this message digest function *digest*.[28]

D3.10. For authentication purposes, for every processor $a \in N$, two cryptographic functions $K_{(a)}$ and $K_{(a)}^{(-1)}$ (which are each other's inverses) are defined in such a way, that it is computationally infeasible to calculate function $K_{(a)}$ from $K_{(a)}^{(-1)}$. Processor a reveals $K_{(a)}^{(-1)}$, but keeps $K_{(a)}$ secret [29].

D3.11. The set of all possible encrypted path parts *EncryptedPathParts* is defined as the smallest set that satisfies:

- $K_{(s)}\{md\ ;\ a\} \in EncryptedPathParts$
  with $a{,}s \in N$ and $md \in MessageDigests,$

and

- $K_{(x)}\{epp; y\} \in EncryptedPathParts$
  with $epp \in EncryptedPathParts$ and $x{,}y \in N$

28. In order to be able to satisfy assumption As2 in Section 3.3.3.4., the message digest function *digest* should be a cryptographic function known by all correct processors with the property that it is computationally infeasible for any processor $p \in N$, which knows a message value mv and its message digest $digest(mv)$, to generate a message value $mv' \neq mv$, such that $digest(mv') = digest(mv)$. In other words, function *digest* should be a ***strong one-way hash function*** [Merk82, Merk89]. See Section 4.1.2.3. for details. Most known message digest functions aim to satisfy this property. For details about message digest functions, see, e.g., [Mull93]
29. An asymmetric cryptosystem, like, e.g., RSA [RiSA78], can be applied to implement the functions $K_{(a)}^{(-1)}$ and $K_{(a)}$. Function $K_{(a)}$ is chosen to be the secret cryptographic function D_a in [RiSA78], whereas $K_{(a)}^{(-1)}$ is chosen to be the corresponding publicly known cryptographic function E_a in [RiSA78]. More details about asymmetric cryptosystems and cryptographic functions can be found in Section 4.1.2.1.

D3.12. The set of all possible valid path parts *ValidPathParts* is defined as the smallest set that satisfies:

$((\underline{\boldsymbol{p}});\ epp) \in ValidPathParts,$

  with $(\underline{\boldsymbol{p}}) \in ValidPaths \wedge epp \in EncryptedPathParts$

and

($(\underline{\boldsymbol{p}}) = (s) \wedge epp = K_{(s)}\{md\ ;\ a\}$

  with $(s;a) \in ValidPaths \wedge md \in MessageDigests$

or

$(\underline{\boldsymbol{p}}) = (\underline{\boldsymbol{q}};\ w;\ x) \wedge epp = K_{(x)}\{pp;\ y\}$

  with $((\underline{\boldsymbol{q}};\ w);\ pp) \in ValidPathParts$

    with $pp = K_{(w)}\{pp';\ x\} \wedge pp \in EncryptedPathParts \wedge$

     $(\underline{\boldsymbol{q}};\ w;\ x;\ y) \in ValidPaths$

)

Informally, for any valid path part $vpp = ((\underline{\boldsymbol{p}});\ epp)$ (with $vpp \in ValidPathParts$, $(\underline{\boldsymbol{p}}) \in ValidPaths$, and $epp \in EncryptedPathParts$), that satisfies definition D3.12, $(\underline{\boldsymbol{p}})$ denotes a path of processors that have subsequently signed the encrypted path part *epp*. This path of processors not strictly necessary, but added for convenience, in order to simplify the decryption of the encrypted path part.

For any path $(\underline{\boldsymbol{p}})$, the function $prefix(\ (\underline{\boldsymbol{p}})\ )$ is defined by:

D3.13. $(\forall\ (\underline{\boldsymbol{p}}) \in Paths :: (empty(\ (\underline{\boldsymbol{p}})\ ) \Rightarrow (\ prefix(\ (\underline{\boldsymbol{p}})\ ) = (\underline{\boldsymbol{p}})\ ))\ )$

$(\forall\ (\underline{\boldsymbol{p}}) \in Paths : \forall\ (\underline{\boldsymbol{q}}) \in Paths : \forall\ a \in N ::$

$(\underline{\boldsymbol{q}};\ a) = (\underline{\boldsymbol{p}}) \Rightarrow (\ prefix(\ (\underline{\boldsymbol{p}})\ ) = (\underline{\boldsymbol{q}})\ )\ )$

Function *messagedigest* is recursively defined by:

D3.14. $(\forall\ vpp \in ValidPathParts : \forall\ (\underline{\boldsymbol{p}}) \in ValidPaths :$

$\forall\ epp \in EncryptedPathParts ::$

$(vpp = ((\underline{\boldsymbol{p}});\ epp)) \Rightarrow$

$(((\exists\ md \in MessageDigests\colon \exists\ s,a \in N :: epp = K_{(s)}\{md\ ;\ a\}) \wedge$

$messagedigest(vpp) = md) \vee$

$((\exists\ epp' \in EncryptedPathParts\colon \exists\ x,y \in N :: epp = K_{(x)}\{epp'\ ,\ y\} \wedge$

$messagedigest(vpp) = messagedigest(prefix((\underline{\boldsymbol{p}})),\ epp')))\ )$

The set of all possible valid messages *ValidMessages* is defined by:

D3.15. $ValidMessages = \{(mv;\ vpp) \mid mv \in MessageValues \wedge vpp \in ValidPathParts \wedge messagedigest(vpp) = digest(mv)\ \}$

For any valid message $vm \in ValidMessages$, where $vm = (mv;\ vpp)$ (with $mv \in MessageValues$ and $vpp \in ValidPathParts$), *mv* is defined as the ***message value*** of *vm*, and *vpp* as the ***path information*** of *vm*.

Assume that, during the multicast process of a BAP, a correct processor $a \in N$ receives a message $mess_i(a) \in Messages$ (with $1 \le i \le M_a$). According to Rq2, processor *a* will only sign message $mess_i$(a) if $mess_i(a)$ is valid. In order to find out whether or not $mess_i(a)$ is a valid message, processor *a* checks if $mess_i(a) \in ValidMessages$. Processor *a* only signs message $mess_i(a)$ if $mess_i(a) \in ValidMessages$. Any correct processor should thus be capable of checking the message structure of received messages (e.g., a valid message should contain a message value followed by a valid path part). In an actual implementation of the protocol, the message structure may, e.g., be indicated by interspersing the various component parts of a message with delimiter characters.

For any valid path part, the function *pathfrompathpart* is recursively defined as follows:

D3.16. $(\forall\ vpp \in ValidPathParts : \forall\ (\mathbf{p}) \in ValidPaths :$
$\forall\ epp \in EncryptedPathParts ::$
$(vpp = ((\mathbf{p}); epp)) \Rightarrow$
$(((\exists\ md \in MessageDigests: \exists\ s,a \in N :: epp = K_{(s)}\{md\ ; a\}) \wedge$
$pathfrompathpart(vpp) = (s;a)) \vee$
$((\exists\ epp' \in EncryptedPathParts: \exists\ x,y \in N :: epp = K_{(x)}\{epp'\ , y\} \wedge$
$pathfrompathpart(vpp)=(pathfrompathpart(prefix((\mathbf{p})), epp'); y)))))$

For any valid message, the functions *messvalue, path, encryptedpathpart,* and *pathofencryptors* are defined by:

D3.17. $(\forall\ vm \in ValidMessages : \forall\ vpp \in ValidPathParts : \forall\ mv \in MessageValues::$
$(vm = (mv\ ; vpp)) \Rightarrow$
$messvalue(vm) = mv \wedge$
$path(vm) = pathfrompathpart(vpp) \wedge$
$(\forall\ (\mathbf{p}) \in Paths : \forall\ epp \in EncryptedPathParts ::$
$(vpp = ((\mathbf{p}); epp)) \Rightarrow$
$encryptedpathpart(vm) = epp \wedge$
$pathofencryptors(vm) = (\mathbf{p})\ )$
$)$

Informally, for any valid message *m*, the above-defined functions *messvalue, path, encryptedpathpart,* and *pathofencryptors* yield the following results:

•	*messvalue*(*m*)	returns the message value of message *m*
•	*path*(*m*)	returns the path described by the path information of message *m*
•	*encryptedpathpart*(*m*)	returns the encrypted path part of the path information of message *m*.
•	*pathofencryptors*(*m*)	returns the path of processors that have subsequently encrypted the path information of message *m*.

Notice that, for any valid message *m* it holds that:

$$pathofencryptors(m) = prefix(path(m))$$

For any message, we define the Boolean predicate *valid* as follows:

D3.18. $(\forall\ m \in Messages :: valid(m) \equiv m \in ValidMessages\ )$

From the definitions given before, it follows that:

$$\forall\ a \in N : \forall\ i \in [1, M_a] :: valid(mess_i(a)) \Rightarrow validpath(path(mess_i(a)))$$

Furthermore, for any message $mess_i(a)$, we define the Boolean predicate $invalid(mess_i(a))$ as follows:

D3.19. $\forall\ a \in N : \forall\ i \in [1, M_a]: invalid(mess_i(a)) \equiv \neg\ valid(mess_i(a))$

In assumptions As3, As4, and As7 in Section 3.3.3.4., the notion of an ***extended path part*** (i.e., a path part concatenated with the identification of a processor) will be applied (Viz. As3, As4, and As7 are assumptions w.r.t. application of the cryptographic functions -defined in D3.10 - on extended path parts). For this purpose we define the following (infinite) set:

D3.20. *ExtendedPathParts* — the set of all possible extended path parts

This set is defined by:
$(\forall\ (\bar{\boldsymbol{e}}) :: (\bar{\boldsymbol{e}}) \in \mathit{ExtendedPathParts} \equiv$
$(\exists\ a \in N : \exists\ \mathit{epp} \in \mathit{EncryptedPathParts} :: ((\bar{\boldsymbol{e}}) = \mathit{epp};\ a)\ ) \vee$
$(\exists\ a \in N : \exists\ \mathit{md} \in \mathit{MessageDigests} :: ((\bar{\boldsymbol{e}}) = \mathit{md};\ a)\ )\ )$

For the proofs in Section 3.3.3.5., we introduce the following definitions.

For any non-empty path $(\underline{\boldsymbol{p}})$, the functions $\mathit{first}(\ (\underline{\boldsymbol{p}})\ )$ respectively $\mathit{last}(\ (\underline{\boldsymbol{p}})\ )$ are defined by:

D3.21. $(\forall\ (\underline{\boldsymbol{p}}) \in \mathit{Paths} : \forall\ (\underline{\boldsymbol{q}}) \in \mathit{Paths} : \forall\ a \in N :: (a;\ \underline{\boldsymbol{q}}) = (\underline{\boldsymbol{p}}) \Rightarrow (\ \mathit{first}(\ (\underline{\boldsymbol{p}})\ ) = a)\ )$
and
$(\forall\ (\underline{\boldsymbol{p}}) \in \mathit{Paths} : \forall\ (\underline{\boldsymbol{q}}) \in \mathit{Paths} : \forall\ a \in N :: (\underline{\boldsymbol{q}};\ a) = (\underline{\boldsymbol{p}}) \Rightarrow (\ \mathit{last}(\ (\underline{\boldsymbol{p}})\ ) = a)\ )$

D3.22. For any processor $a \in N$, two Boolean predicates are introduced, *correct*(a) and *faulty*(a) respectively. The predicate *correct*(a) holds iff processor a is correct, whereas the predicate *faulty*(a) holds iff processor a is faulty. Notice that this implies that:
$(\forall\ a \in N :: (\mathit{correct}(a) \Rightarrow \neg\ \mathit{faulty}(a)) \wedge (\neg\ \mathit{faulty}(a) \Rightarrow \mathit{correct}(a))\ )$

We introduce the following notation:

N3.2. Let $\mathit{mess}_i(a)$ be the i-th message that processor a receives $(1 \leq i \leq M_a)$. Assume that $\mathit{mess}_i(a) \in \mathit{ValidMessages}$, with $\mathit{messvalue}(\mathit{mess}_i(a)) = \mathit{mv}$, and $\mathit{path}(\mathit{mess}_i(a)) = (\underline{\boldsymbol{p}})$. Such a valid message will be denoted as $m(\mathit{mv}, (\underline{\boldsymbol{p}}))$.

3.3.3.2. Generating a message

The message generation process is performed by any correct source, say s ($s \in N$). Roughly, this message generation process consists of the following steps:

Gs1. The source s generates a message value $\mathit{mv} \in \mathit{MessageValues}$, which it wants to communicate to all destination processors.

Gs2. For any processor $a \in N$ $(a \neq s)$, which s wants to send a message to, s generates a valid message $\mathit{vm} \in \mathit{ValidMessages}$, where $\mathit{vm} = (\mathit{mv};\ \mathit{vpp})$, with $\mathit{vpp} = ((s);\ K_{(s)}\ \{\mathit{digest}(\mathit{mv})\ ;\ a\})$, and sends vm to a. Notice that, according to N3.2, $\mathit{vm} = m(\mathit{mv}, (s;a))$.

3.3.3.3. Relaying a message

Roughly, the message relay process as it is performed by any correct processor $a \in N$ consists of the following steps:

Rs1. At receipt of a message, processor a checks if the received message is valid or not.

Rs2. If the received message is not valid, it is rejected by a.

Rs3. If the received message is valid, but received too late, not expected, or not received for the first time[30], it is also rejected by *a*.

Rs4. If the received message is valid, and it is timely received, it was expected, and it is received for the first time, it is relayed by *a* in such a way that the resulting message is again a valid message. Any correct processor $a \in N$, that wants to relay a valid message $vm \in ValidMessages$ to a certain processor $b \in N$ (with $validpath(path(vm); b)$), sends $relay_{(a,b)}(vm)$ to *b* (where function *relay* is defined below).

Function *relay* mentioned in step Rs4 is formally defined as follows.

Assume that a correct processor $a \in N$ wants to relay a valid message $vm \in ValidMessages$ to a certain processor $b \in N$ (with $validpath(path(vm); b)$), then it performs the function $relay_{(a,b)}(vm)$ which is defined by:

D3.23. $(\forall\ vm \in ValidMessages: \forall\ mv \in MessageValues:$
$$\forall\ (\underline{\mathbf{p}}) \in ValidPaths: \forall\ epp \in EncryptedPathParts ::$$
$$(vm = (mv; ((\underline{\mathbf{p}}); epp)) \wedge validpath(path(vm); b)) \Rightarrow$$
$$relay_{(a,b)}(vm) = (mv; ((\underline{\mathbf{p}}; a); K_{(a)}\{epp; b\})))$$

Notice that the function *relay* can also be defined by:

D3.24. $(\forall\ a \in N : \forall\ mv \in MessageValues: \forall\ (\underline{\mathbf{p}}) \in ValidPaths::$
$$length((\underline{\mathbf{p}})) \geq 2 \Rightarrow relay_{(last((\underline{\mathbf{p}})),a)}\ (m(mv, (\underline{\mathbf{p}}))) = m(mv, (\underline{\mathbf{p}}; a)))$$

Notice that, if a processor $a \in N$ performs the function *relay* on a valid message *m*, then, implicitly, processor *a* signs *m* (i.e., applies its secret cryptographic function $K_{(a)}$ on the extended path part of *m*).

3.3.3.4. Assumptions with regard to the authentication of messages

In this section, we formulate the following so-called ***authentication assumptions***, i.e., assumptions with regard to the authentication of messages. Following the list of assumptions, an indication is given how these assumptions can be satisfied.

As1. $(\forall\ m \in ValidMessages ::$
$$(\exists\ mv \in MessageValues: \exists\ vpp \in ValidPathParts ::$$
$$m = (mv; vpp) \wedge messagedigest(vpp) = digest(mv)))$$
i.e., the message digest of the message value integrated in the path information of a valid message is compatible with the message value of that message.

30. More precise definitions of valid messages that are received too late, that are not expected, or not received for the first time are given in Section 3.3.4., Section 3.3.7., and Section 3.3.6. and 3.3.7. respectively.

As will turn out later, such valid messages must have been created by a faulty processor. Such valid messages are rejected by a correct processor, since *acceptance* of one or more of these messages by a correct processor may lead to violation of the interactive consistency conditions. Notice that *rejection* of messages created by faulty processors does not have a negative effect on satisfaction of the interactive consistency conditions (i.e., it is impossible that a BAP B_1 in which a message *m* created by a faulty processor is accepted **does** satisfy the interactive consistency conditions, while a BAP B_2 which is identical to BAP B_1 with the mere difference that message *m* is rejected in B_2 (while *m* is accepted in B_1), **does not** satisfy the interactive consistency conditions).

As2. $(\forall\ mv, mv' \in MessageValues :: (digest(mv) = digest(mv')) \Rightarrow (mv = mv'))$
i.e., a message value is uniquely determined by its message digest.

As3. $(\forall\ a,b \in N : \forall\ (\bar{e}) \in ExtendedPathParts :: (a \neq b\ ) \Rightarrow K_{(a)}\{\ (\bar{e})\ \} \neq K_{(b)}\{\ (\bar{e})\ \})$
i.e., encryption of an extended path part with the secret cryptographic functions of different processors leads to different results.

As4. $(\forall\ (\bar{e}) \in ExtendedPathParts : \forall\ a \in N :: (\bar{e}) = K_{(a)}^{(-1)}\ \{\ K_{(a)}\{(\bar{e})\}\}$
i.e., for every processor, its cryptographic functions are each other's inverses.

As5. $(\forall\ a,b \in N : (a \neq b) \wedge correct(b) \Rightarrow a$ knows $K_{(b)}^{(-1)}$ but not $K_{(b)})$
i.e., every processor a knows the public cryptographic function of every correct processor b (with $b \neq a$), but not b's secret cryptographic function.

As6. $(\forall\ a \in N : a$ knows $K_{(a)}$ and $K_{(a)}^{(-1)})$
i.e., every processor knows its own secret and public cryptographic function.

As7. $(\forall\ a \in N : \forall\ (\bar{e}), (\bar{f}) \in ExtendedPathParts :: ((\bar{e}) \neq (\bar{f})) \Rightarrow K_{(a)}\{\ (\bar{e})\ \} \neq K_{(a)}\{\ (\bar{f})\ \})$
i.e., for any processor, encryption of different extended path parts with the secret cryptographic functions of this processor leads to different results.

As8. Function *digest* is chosen such that it is computationally infeasible to find or apply the inverse function $digest^{(-1)}$.

Assumption As1 expresses that the message digest of the message value integrated in the path information of a valid message should be compatible with the message value of that message. In our BAPs, this assumption can easily be satisfied by having a correct source generate message according to the message generation process described in Section 3.3.3.2., and by having correct destination processors not alter the message value of a message while relaying it. In the message relay process as described in Section 3.3.3.3., the latter preliminary is met.

Notice that, in order to be able to satisfy assumption As2 and As8, the message digest function *digest* should be a cryptographic function known by all correct processors with the property that it is computationally infeasible for any processor $p \in N$, which knows a certain (arbitrary) message value *mv* and its message digest *digest*(*mv*), to generate a message value $mv' \neq mv$, such that $digest(mv') = digest(mv)$. In other words, function *digest* should be a ***strong one-way hash function*** [Merk82, Merk89]. See Section 4.1.2.3. for details. Most known message digest functions aim to satisfy this property. For details about message digest functions, see, e.g., [Mull93].

As3, As5, and As6 express the assumptions that a correct processor's signature is unforgeable, that the authenticity of messages signed by a correct processor $c \in N$ can be verified by any processor $\in N$ other than c. Assumption As4 expresses that the secret and public cryptographic function of every processor $a \in N$ are each other's inverses. These assumptions As3 through As6 can be satisfied by applying cryptographic techniques as discussed in Section 3.1.4.2.

Assumption As7 expresses that encryption of any extended path part yields a unique encrypted path part. This means that the secret cryptographic functions must be bijective. The fact that every secret cryptographic function has an inverse function (D3.10, As4) already implies satisfaction of this requirement.

In the following section, we prove that the unforgeability properties UP1 and UP2 are satisfied. The proof relies on the following two assumptions, which are not explicitly stated in the proof:

1. Application of a cryptographic function other than $K_{(c)}$ on an **invalid** message does not yield a result which is equal to a valid message encrypted with secret cryptographic function $K_{(c)}$.
2. Processor f cannot re-use previously collected valid messages signed by processor c in order to obtain the desired message.
   Each time that a BAP is being executed, processor f receives a number of valid messages during the multicast process of that BAP. Processor f may also obtain valid messages from other faulty processors by colluding with them. Any malicious processor may collect (some or all of) these messages in order to re-use them during execution of a subsequent BAP. However, as long as the collection of messages possessed by processor f does not contain the desired message m, it is impossible for processor f to generate m from the set of messages that f has collected. The latter can be guaranteed by requiring every valid message to be unique, i.e., every valid message should contain a unique identification (e.g., a BAP-number which is unique for each BAP that is executed). By including such a unique identification in the signed path information of every valid message, malicious processors can be prevented from re-using previously collected valid messages.

3.3.3.5. Satisfaction of the unforgeability properties

In this section, we will recall the definition of the unforgeability properties, and prove that these unforgeability properties hold, provided that assumptions As1 though As8 hold.

Recall that the unforgeability properties UP1 and UP2 have been defined as follows:

UP1	If a processor d receives a valid message $m(mv, (s; \underline{\mathbf{p}}; c; \underline{\mathbf{q}}; d))$, where c is the identification of a correct processor, then processor c has received and accepted a valid message $m(mv, (s; \underline{\mathbf{p}}; c))$.
UP2	If a processor d receives a valid message $m(mv, (s; \underline{\mathbf{q}}; d))$, where s is the identification of a correct processor, then processor s has generated and multicast a valid message $m(mv, (s;x))$ to the selected processors x in the system, and it has accepted a copy of the message itself.

THEOREM 3.4.
Provided that assumptions As1 through As8 are satisfied, the unforgeability properties UP1 and UP2 hold.

Proof:
In order to prove satisfaction of the unforgeability properties UP1 and UP2, the con-

struction of valid messages will be described in an alternative way. In order to give this alternative description, we will redefine the message relay function.

Recall the definition of function *relay* as expressed in definition D3.23 :

$$\begin{aligned}&(\forall\ vm \in \mathit{ValidMessages}: \forall\ mv \in \mathit{MessageValues}:\\ &\quad \forall\ (\underline{\boldsymbol{p}}) \in \mathit{ValidPaths}: \forall\ epp \in \mathit{EncryptedPathParts} ::\\ &\qquad (vm = (mv; ((\underline{\boldsymbol{p}}); epp)) \wedge \mathit{validpath}(\mathit{path}(vm); b)) \Rightarrow\\ &\qquad \mathit{relay}_{(a,b)}(vm) = (mv; ((\underline{\boldsymbol{p}}; a); K_{(a)}\{epp; b\}))\ )\end{aligned} \tag{1}$$

If we extend the definition of *EncryptedPathParts* to also contain the set of message digests *MessageDigests*, then, in fact, in the message generation process, any correct source *s* applies function *relay* on some valid message (*mv*; ((); *digest*(*mv*))), with $mv \in \mathit{MessageValues}$. Hence, for any correct source *s* that generates a valid message *m*(*mv*,(*s*)) (with message value *mv*) and relays it to some processor $a \in N$, processor *a* will receive from *s* the following message:

$$\mathit{relay}_{(s,a)}((mv; ((); \mathit{digest}(mv)))) = (mv; ((s); K_{(s)}\{\mathit{digest}(mv); a\})) \tag{2}$$

Notice that, by As4, function $\mathit{relay}_{(a,b)}$ (with $a,b \in N$) is invertible, i.e., function $\mathit{relay}_{(a,b)}^{(-1)}$ exists.

For any processors $a, b \in N$, and any valid message $vm \in \mathit{ValidMessages}$, with *vm* = (*mv*; ((***p***); *epp*)) and *validpath*(*path*(*vm*);*b*)), function *relay* can also be expressed as

$$\mathit{relay}_{(a,b)}((mv; ((\underline{\boldsymbol{p}}); epp))) = (mv; ((\underline{\boldsymbol{p}};a); R_{(a,b)}(epp))) \tag{3}$$

in which

$$R_{(a,b)}(epp) = K_{(a)}\{(epp;b)\} \tag{4}$$

Clearly, function $R_{(a,b)}$ is invertible, i.e., function $R_{(a,b)}^{(-1)}$ exists.

Using (1) and (2), respectively, a valid message *m*(*mv*,(***p***)), with $(\underline{\boldsymbol{p}}) = (p_1;p_2;\ldots;p_v)$ (for $2 \le v \le T+2$) can be expressed by

$$\mathit{relay}_{(p_{v-1},p_v)}(\mathit{relay}_{(p_{v-2},p_{v-1})}(\ldots(\mathit{relay}_{(p_1,p_2)}((mv; ((); \mathit{digest}(mv)))))\ldots)) \tag{5}$$

and, using (3), by

$$(mv; (\mathit{prefix}((\underline{\boldsymbol{p}})); (R_{(p_{v-1},p_v)}(R_{(p_{v-2},p_{v-1})}(\ldots(R_{(p_1,p_2)}(\mathit{digest}(mv)))\ldots))))) \tag{6}$$

By defining, for any valid path (***p***) (with $\mathit{length}((\underline{\boldsymbol{p}})) \ge 2$)

$$R_{((\underline{\boldsymbol{p}}))} = R_{(p_{v-1},p_v)} \circ R_{(p_{v-2},p_{v-1})} \circ \ldots \circ R_{(p_1,p_2)} \tag{7}$$

and, for any valid path (***p***) (with $\mathit{length}((\underline{\boldsymbol{p}})) = 1$)

$$R_{((\underline{\boldsymbol{p}}))} = I \tag{8}$$

(in which *I* is the identity function), message *m*(*mv*,(***p***)) (with $\mathit{length}((\underline{\boldsymbol{p}})) \ge 1$) can be expressed by

$$m(mv,(\underline{\boldsymbol{p}})) = (mv; (\mathit{prefix}((\underline{\boldsymbol{p}})); R_{((\underline{\boldsymbol{p}}))}(\mathit{digest}(mv)))) \tag{9}$$

In a similar way, a valid message $m(mv, (s;\underline{\boldsymbol{p}};c;\underline{\boldsymbol{q}};d))$ can be expressed as

$$(mv;\ ((s;\underline{\boldsymbol{p}};c;\underline{\boldsymbol{q}})\ ;\ (R_{((\underline{\boldsymbol{q}};d))}{}^{\circ}R_{(c,first((\underline{\boldsymbol{q}};d))}{}^{\circ}R_{((s;\underline{\boldsymbol{p}}))}(digest(mv))))) \tag{10}$$

Again, notice that for any $(\underline{\boldsymbol{p}}) \in$ *Validpaths*, $R_{((\underline{\boldsymbol{p}}))}$ is invertible, i.e., $R_{((\underline{\boldsymbol{p}}))}{}^{(-1)}$ exists.

By describing valid messages in the alternative way presented above, we are able to prove that the unforgeability properties UP1 and UP2 hold.

We first prove satisfaction of unforgeability property UP1.

Assume that a processor d receives a valid message $m(mv,(s;\underline{\boldsymbol{p}};c;\underline{\boldsymbol{q}};d))$, where c is the identification of a correct processor. Then, we must prove that c has received and accepted a valid message $m(mv,(s;\underline{\boldsymbol{p}};c))$.

By (4) and (10), message $m(mv,(s;\underline{\boldsymbol{p}};c;\underline{\boldsymbol{q}};d))$ can be expressed as

$$(mv;\ ((s;\underline{\boldsymbol{p}};c;\underline{\boldsymbol{q}})\ ;\ (R_{((\underline{\boldsymbol{q}};d))}(K_{(c)}\{R_{((s;\underline{\boldsymbol{p}}))}(digest(mv));\ first((\underline{\boldsymbol{q}};d))\}\ )))) \tag{11}$$

For any $a \in N$, a concatenation function $C_{(a)}$ is defined on any encrypted path path *epp* $\in$ *EncryptedPathParts* by

$$C_{(a)}(epp) = epp;a \tag{12}$$

By (11) and (12), we are able to express message $m(mv,(s;\underline{\boldsymbol{p}};c;\underline{\boldsymbol{q}};d))$ as

$$(mv;\ ((s;\underline{\boldsymbol{p}};c;\underline{\boldsymbol{q}})\ ;\ (R_{((\underline{\boldsymbol{q}};d))}(K_{(c)}\{C_{(first((\underline{\boldsymbol{q}};d)))}(R_{((s;\underline{\boldsymbol{p}}))}(digest(mv)))\}\ )))) \tag{13}$$

Message $m(mv,(s;\underline{\boldsymbol{p}};c;\underline{\boldsymbol{q}};d))$ consists of a message value concatenated with a valid path part. This valid path part, in its turn, consists of a path of encryptors concatenated with an encrypted path part. The path of encryptors (in this case $(s;\underline{\boldsymbol{p}};c;\underline{\boldsymbol{q}})$) does not influence the proof, for convenience, it has been omitted in the reasoning below. Thus, for convenience, message $m(mv,(s;\underline{\boldsymbol{p}};c;\underline{\boldsymbol{q}};d))$ is described as a tuple

$$(mv;\ (R_{((\underline{\boldsymbol{q}};d))}(K_{(c)}\{C_{(first((\underline{\boldsymbol{q}};d)))}(R_{((s;\underline{\boldsymbol{p}}))}(digest(mv)))\}\ ))) \tag{14}$$

Hence, we should consider all possible ways in which message $m(mv,(s;\underline{\boldsymbol{p}};c;\underline{\boldsymbol{q}};d))$ can be generated.

Tuples of the form

$$(mv;\ (R_{((\underline{\boldsymbol{q}};d))}(K_{(c)}\{C_{(first((\underline{\boldsymbol{q}};d)))}(R_{((s;\underline{\boldsymbol{p}}))}(digest(mv)))\}\ )))$$

can only be generated in the following six ways:

W1. $(mv\ ;\ (R_{((\underline{\boldsymbol{q}};d))}(K_{(c)}\{C_{(first((\underline{\boldsymbol{q}};d)))}(R_{((s;\underline{\boldsymbol{p}}))}(digest(mv)))\}\ )))$

W2. $(digest^{(-1)}(mv)\ ;\ (R_{((\underline{\boldsymbol{q}};d))}(K_{(c)}\{C_{(first((\underline{\boldsymbol{q}};d)))}(R_{((s;\underline{\boldsymbol{p}}))}(mv))\}\ )))$

W3. $(digest^{(-1)}(R_{((s;\underline{\boldsymbol{p}}))}{}^{(-1)}(mv))\ ;\ (R_{((\underline{\boldsymbol{q}};d))}(K_{(c)}\{C_{(first((\underline{\boldsymbol{q}};d)))}(mv)\}\ )))$

W4. $(digest^{(-1)}(R_{((s;\underline{\boldsymbol{p}}))}{}^{(-1)}(C_{(first((\underline{\boldsymbol{q}};d)))}{}^{(-1)}(mv)))\ ;\ (R_{((\underline{\boldsymbol{q}};d))}(K_{(c)}\{mv\}\ )))$

W5. $(digest^{(-1)}(R_{((s;\underline{\mathbf{p}}))}{}^{(-1)}(C_{(first((\underline{\mathbf{q}};d)))}{}^{(-1)}(K_{(c)}{}^{(-1)}\{mv\}))) \,;\, (R_{((\underline{\mathbf{q}};d))}(mv)))$

W6. $(digest^{(-1)}(R_{((s;\underline{\mathbf{p}}))}{}^{(-1)}(C_{(first((\underline{\mathbf{q}};d)))}{}^{(-1)}(K_{(c)}{}^{(-1)}\{R_{((\underline{\mathbf{q}};d))}{}^{(-1)}(mv)\}))) \,;\, (mv))$

We will now show that W1 is the only way in which message $m(mv,(s;\underline{\mathbf{p}};c;\underline{\mathbf{q}};d))$ can be generated.

In W2 through W6, application of function $digest^{(-1)}$ or of a function which is the composition of $digest^{(-1)}$ and an invertible function is required. However, since function *digest* is a strong one-way hash function, by As8, it is computationally infeasible to find or apply function $digest^{(-1)}$. This implies that it is also computationally infeasible to find or apply the composition of $digest^{(-1)}$ and an invertible function. This can be seen as follows.

Assume that it is computationally infeasible to find or apply $digest^{(-1)}$ but the composition of $digest^{(-1)}$ and an invertible function, say *F*, can be easily found. However, since *F* is an invertible function, $F^{(-1)}$ can be found, and hence the composition of $F^{(-1)}$ and $F \circ digest^{(-1)}$ can also be found. However, since $F^{(-1)} \circ F = I$ (the identity function) this would imply that $digest^{(-1)}$ can be found, which contradicts the assumption. Thus, it is computationally infeasible to find or apply the composition of $digest^{(-1)}$ and an invertible function.

Hence, we conclude that W1 is the only way in which message $m(mv,(s;\underline{\mathbf{p}};c;\underline{\mathbf{q}};d))$ can be generated.

Because processor *c* is correct, by As5 and As6, the cryptographic function $K_{(c)}$ is only known by processor *c*, and thus, function $K_{(c)}$ can only be applied by processor *c*. Moreover, in order to produce the result

$$K_{(c)}\{C_{(first((\underline{\mathbf{q}};d)))}(R_{((s;\underline{\mathbf{p}}))}(digest(mv)))\}$$

by As3 and As7, processor *c* must possess

$$R_{((s;\underline{\mathbf{p}}))}(digest(mv))$$

Since any correct processor only signs valid messages, by As1 and As2, processor *c* must have received and accepted a valid message

$$m(mv,(s;\underline{\mathbf{q}};c)) = (mv;\ (s;\underline{\mathbf{q}});\ R_{((s;\underline{\mathbf{p}}))}(digest(mv)))$$

This proves unforgeability property UP1.

Now we prove unforgeability property UP2.

Assume that a processor *d* receives a valid message $m(mv,(s;\underline{\mathbf{q}};d))$, where *s* is the identification of a correct processor. Then, we must prove that *s* has generated and multicast a message $m(mv,(s;x))$ to the selected processors *x* in the system, and it has accepted a copy of the message itself.

By (4) and (10), message $m(mv,(s;\underline{\mathbf{q}};d))$ can be expressed as

$$(mv;\ ((s;\underline{\mathbf{q}})\ ;\ (R_{\,((\underline{\mathbf{q}};d))}(K_{\,(s)}\{digest(mv));\ first((\underline{\mathbf{q}};d))\}\ )))) \qquad (15)$$

By (12) and (15), we are able to express message $m(mv,(s;\underline{\mathbf{q}};d))$ as

$$(mv;\ ((s;\underline{\mathbf{q}})\ ;\ (R_{\,((\underline{\mathbf{q}};d))}(K_{\,(s)}\{C_{(first((\underline{\mathbf{q}};d)))}(digest(mv))\}\ )))) \qquad (16)$$

Message $m(mv,(s;\underline{\mathbf{q}};d))$ consists of a message value concatenated with a valid path part. This valid path part, in its turn, consists of a path of encryptors concatenated with an encrypted path part. The path of encryptors (in this case $(s;\underline{\mathbf{q}})$) does not influence the proof, for convenience, it has been omitted in the reasoning below. Thus, for convenience, message $m(mv,(s;\underline{\mathbf{q}};d))$ is described as a tuple

$$(mv;\ (R_{\,((\underline{\mathbf{q}};d))}(K_{\,(s)}\{C_{(first((\underline{\mathbf{q}};d)))}(digest(mv))\}\ ))) \qquad (17)$$

Hence, we should consider all possible ways in which message $m(mv,(s;\underline{\mathbf{q}};d))$ can be generated.

Tuples of the form

$$(mv;\ (R_{\,((\underline{\mathbf{q}};d))}(K_{\,(s)}\{C_{(first((\underline{\mathbf{q}};d)))}(digest(mv))\}\ )))$$

can only be generated in the following five ways:

W1. $(mv\ ;\ (R_{\,((\underline{\mathbf{q}};d))}(K_{\,(s)}\{C_{(first((\underline{\mathbf{q}};d)))}(digest(mv))\}\ )))$

W2. $(digest^{(-1)}(mv)\ ;\ (R_{\,((\underline{\mathbf{q}};d))}(K_{\,(s)}\{C_{(first((\underline{\mathbf{q}};d)))}(mv)\}\ )))$

W3. $(digest^{(-1)}(C_{(first((\underline{\mathbf{q}};d)))}{}^{(-1)}(mv))\ ;\ (R_{\,((\underline{\mathbf{q}};d))}(K_{\,(s)}\{mv\}\ )))$

W4. $(digest^{(-1)}(C_{(first((\underline{\mathbf{q}};d)))}{}^{(-1)}(K_{\,(s)}{}^{(-1)}\{mv\}))\ ;\ (R_{\,((\underline{\mathbf{q}};d))}(mv)))$

W5. $(digest^{(-1)}(C_{(first((\underline{\mathbf{q}};d)))}{}^{(-1)}(K_{\,(s)}{}^{(-1)}\{R_{\,((\underline{\mathbf{q}};d))}{}^{(-1)}(mv)\}))\ ;\ (mv))$

We will now show that W1 is the only way in which message $m(mv,(s;\underline{\mathbf{q}};d))$ can be generated.

In W2 through W5, application of function $digest^{(-1)}$ or of a function which is the composition of $digest^{(-1)}$ and an invertible function is required. However, since function *digest* is a strong one-way hash function, by As8, it is computationally infeasible to find or apply function $digest^{(-1)}$. As shown in the proof of unforgeability property UP1, this implies that it is also computationally infeasible to find or apply the composition of $digest^{(-1)}$ and an invertible function.

Hence, we conclude that W1 is the only way in which message $m(mv,(s;\underline{\mathbf{q}};d))$ can be generated.

Because processor *s* is correct, by As5 and As6, the cryptographic function $K_{\,(s)}$ is only known by processor *s*, and thus, function $K_{\,(s)}$ can only be applied by processor *s*. Thus, processor *s* must have produced the result

$$K_{\,(s)}\{C_{(first((\underline{\mathbf{q}};d)))}(digest(mv))\}$$

and hence, processor *s* must have generated a valid message $m(mv,(s;first((\underline{\mathbf{q}};d))))$.

Since processor *s* is correct, *s* must be the source of a BAP in which this message has been produced, and hence, *s* must have generated and multicast a message $m(mv,(s;x))$ to the selected processors *x* in the system, and it must have accepted a copy of the message itself.

This proves unforgeability property UP2.

Thus, provided that assumptions As1 through As8 hold, the unforgeability properties UP1 and UP2 hold. This proves Theorem 3.4. ❑

3.3.3.6. Final remarks on message authentication

In a lock-step synchronous *fully-connected* system, the receiver of a message knows which processor sent it, and furthermore, the path along which the message travelled may be deduced from the time slot in which the message arrives in the destination processor. In such a system, it is sufficient that messages are *only signed by the source* (and not by destination processors that relay messages).

However, as discussed before, in a fully-connected synchronous system with arbitrary nonnegative ρ and τ_{max}, and arbitrary nonnegative $\tau_{min} \le \tau_{max}$, the path along which a message travelled from the source to a destination processor can *not* be deduced from the time slot in which the message arrives in the destination processor. The required information is obtained by including path information in each message. Then, for the unforgeability properties UP1 and UP2 to hold, it is required that *every correct processor signs the path information of every valid message it relays*.

Nevertheless, in many authenticated BAPs for lock-step synchronous systems, it is also assumed that every correct processor signs every message it relays, for this makes the protocols also applicable for lock-step synchronous systems that are not fully-connected[31]. In such systems, there may be no direct communication link between sender and receiver of a message, so in this case, the signature of the sender is required to enable the receiver to determine the sender of the message.

In a similar way, since the authenticated self-synchronizing BAPs described in this section are also based on the assumption that every correct processor signs every message it relays, the authenticated self-synchronizing BAPs also work in a synchronous system which is not fully-connected. In such a system, τ_{max} is defined as the real-time upper bound on the time needed to communicate a message from the sender (directly or via other processors) to the receiver of the message.

Notice that the message syntax proposed in this section inhibits application of the third reduction strategy described in Section 3.2. (i.e., reduction of the size of messages by encoding messages into symbols of an erasure-correcting code). In the protocols in Section 3.2., in each communication phase, the message value of any received valid message is encoded into symbols of an erasure-correcting code, and these symbols form the message values of the messages that are relayed. However, by definition

31. Lamport et al. in [LaSP82] show that the system need not always be fully-connected in order to guarantee Byzantine Agreement. They prove that the connectivity of the system must be at least T+1 in order to be able to guarantee Byzantine Agreement.

D3.19, the message digest of the message value integrated in the path information is the result of application of the message digest function *digest* on the message value of the message originally sent by the *source*. Thus, in an authenticated BAP based on encoding messages into symbols of an erasure-correcting code, the message digest of the message value integrated in the path information of a valid message *m* (i.e., the result of application of the message digest function *digest* on the message value of the message originally sent by the source) will in general be incompatible[32] with the message digest of the message value of *m* during the multicast process of the BAP (i.e., the result of application of the message digest function *digest* on a symbol of an erasure-correcting code). This violates assumption As1, which states that the message digest of the message value integrated in the path information of a message should be compatible with the message value of that message.

3.3.4. Possible fault behaviour of faulty processors

Having observed that message authentication limits the possible fault behaviour of faulty processors during the multicast process of an authenticated self-synchronizing BAP, it is interesting to determine the exact fault behaviour that may be exhibited by faulty processors during the multicast process of that BAP, if compared to correct processors.

A multicast process of an authenticated self-synchronizing BAP consists of a message generation process (see Section 3.3.3.2.), and several message relay processes (see Section 3.3.3.3.). In the message generation process, a source generates a message and sends it to a number of other processors in the system. In a message relay process, a processor that receives a message, relays it to a number of other processors in the system.

We will first consider possible fault behaviour of a faulty source in the *message generation process*. Then, the possible fault behaviour of a processor that receives a valid message in a *message relay process*, will be considered.

After having described the *possible* fault behaviour of faulty processors during the message generation process and the message relay process respectively, it is possible to describe the *observed* behaviour (either faulty or correct) of a faulty processor. We have implemented a simulation program of an authenticated self-synchronizing BAP (see [Post96] for details), in which the observed behaviour of the faulty processors is as described below. This simulation program has been used in order to experimentally verify if the authenticated self-synchronizing BAPs described in this section are indeed resilient to up to *T* faulty processors that exhibit the behaviour as described below.

Possible fault behaviour during the message generation process
If the source (in the *message generation process*) is correct, then it will generate a valid message, and send it in time (see D3.27. in the next section) to each processor in the subset of processors to which it should send a message, according to the BAP.

A faulty source may exhibit the following behaviour:

❏ it may refuse to generate a message.

32. Unless only repetition codes are used.

- ❑ it may collude with other faulty processors in order to generate a message that has been signed with the signatures of one or more faulty processors. For the message to be valid, the identification of the sender and the receiver must be the last two identifications in the path described in the path information of the message.
- ❑ it may generate messages with arbitrary message values.
- ❑ it may send one or more messages to some processors but not to others.
- ❑ it may send messages at arbitrary time instances. This may only affect the decision-making process, if the message arrives before the end of the multicast process of the BAP, otherwise it is indistinguishable from refusing to send a message.
- ❑ it may send an invalid message. However, since all invalid messages will be rejected, this behaviour is indistinguishable from refusing to send a message.

The *observed* behaviour of a faulty source during the message generation process is as follows:

- ❑ the source generates several valid messages with arbitrary message values and path information describing a path containing the identifications of one or more faulty processors, where the identification of the source and the receiver are the last two identifications in the path.
- ❑ a number of these generated messages (possibly zero) are sent to one or more processors at arbitrary time instances.

Possible fault behaviour during the message relay process

If a processor that receives a valid message (in the *message relay process*) is correct, then it will relay this message and send it in time (see D3.27. in the next section) to each processor in the subset of processors to which it should send a message, according to the BAP.

A faulty processor $f \in N$ that should relay a received valid message may exhibit the following behaviour:

- ❑ it may refuse to relay the message.
- ❑ it may collude with other faulty processors in order to sign the received message with the signatures of one or more faulty processors, the identification of which is not yet in the path described by the path information of the received message. For the message to be valid, f's identification and the identification of the receiver must be the last two identifications in the path described in the path information of the message.
- ❑ it may relay the message one or more times to some processors but not to others.
- ❑ it may relay the message at arbitrary time instances. This may only affect the decision-making process, if the message arrives before the end of the multicast process of the BAP, otherwise it is indistinguishable from refusing to relay a message.
- ❑ it may corrupt the message (resulting in an invalid message) before relaying it. However, since all invalid messages will be rejected, this behaviour is indistinguishable from refusing to send a message.

So, the *observed* behaviour of a faulty processor $f \in N$ that should relay a received valid message during the message relay process is as follows:

- ❑ processor *f* generates a number of messages by signing the received message with the signatures of one or more faulty processors, the identification of which is not yet in the path described by the path information of the received message. The signature of processor *f* itself is the last one that is applied to the message.
- ❑ a number of these generated messages (possibly zero) are relayed to one or more processors at arbitrary time instances.

3.3.5. Definitions with regard to the timeliness of valid messages

In this section, we use the following definitions:

D3.25. A valid message *m* is ***received in time*** in a correct processor *c* if it arrives in *c* in or before communication phase *i* ($1 \leq i \leq T+1$) of *c*'s sub-BAP, and the path information of *m* describes a path containing at least *i*+1 processors.

D3.26. A valid message *m* is ***received too late*** in a correct processor *c* if it arrives in *c* after *c* has concluded communication phase *i* ($1 \leq i \leq T+1$) of its sub-BAP, and the path information of *m* describes a path containing at most *i*+1 processors.

D3.27. A valid message *m*, sent directly from the source to a destination processor *d* is ***sent in time***, if, after the source started the BAP (by sending the first valid message of the BAP), *m* arrives in *d* within a real-time interval of length between τ_{min} and τ_{max}. Similarly, a valid message *m*, relayed by a destination processor *j* in communication phase *i* of its sub-BAP to another destination processor *k* is ***sent in time*** if *m* arrives in *k* within a real-time interval of length between τ_{min} and τ_{max} after *j* started communication phase *i* of its sub-BAP.

Notice that, by definitions D3.25 and D3.27 respectively, all valid messages that ***may be*** received (sent) in time, are defined as being received (sent) in time. By D3.26, a valid message is only defined as being received too late, if the receiving processor is sure that the message cannot have been sent in time. The reason that we choose the definitions this way, is that, since the correct processors do not know the start of the BAP, there is no possibility for a correct processor to always determine whether a valid message is in time or too late. Acceptance of a valid message that was, in reality, received too late, but was received in time according to definition D3.25, does not influence the result of the BAP[33]. However, rejection of a valid message that was indeed sent in time may lead to violation of interactive consistency condition IC2, and thus inhibit Byzantine Agreement. By choosing appropriate lengths of the various communication phases of our BAPs, we make sure that all valid messages that, according to definitions D3.25 and D3.27, are received (sent) in time are accepted by all correct processors before the end of the BAP. This is proved in Theorem 3.5. through 3.10. respectively.

In self-synchronizing BAPs, processors do not a priori know when a BAP starts. Therefore, it is not guaranteed that each communication phase begins and ends simultaneously in each processor. All processors may start and end any communication

33. See Section 3.3.9. for more details.

phase i at different times. Therefore, for every i and every pair of *correct*[34] processors p and q, we clearly distinguish between communication phase i in processor p and communication phase i in q.

3.3.6. A description of an authenticated self-synchronizing BAP

Before we give the description of the authenticated self-synchronizing BAP, we make the following remarks.

- Assumption A3.4 (in Section 3.3.8.) states that a processor may concurrently execute different BAPs. Without further specification, it is assumed that messages of different BAPs can be distinguished, such that every correct processor can determine when it receives the first valid message of a new BAP. This implies that, given the contents of any received valid message, any correct processor should be able to uniquely identify the BAP to which this message belongs.

  In fact, a processor may concurrently execute different ICAs, each of which consists of a number of concurrently executed BAPs. So, in order to identify the BAP to which a received message belongs, every message in every BAP should carry an identification which states to which BAP of which ICA the message belongs.

  To *identify the ICA* to which a received message belongs, every ICA can be given a unique identification, e.g., a sequence number. Every ICA consists of execution of a number of BAPs.

  Since all BAPs belonging to a certain ICA have a unique source, the source of a received valid message uniquely *identifies the BAP* (within the identified ICA) to which that message belongs. The source processor of a certain BAP is the first processor in the path described by the path information of any valid message belonging to that BAP. So, the path information of a valid message can be used to identify the BAP (within the identified ICA) to which that message belongs.

  To avoid confusion, in the following description of an authenticated self-synchronizing BAP, we will omit the process of identifying, for each received valid message, the BAP and the ICA to which that message belongs, since this would only unnecessarily complicate the understanding of how the protocol works.

- In order to be able to decide whether or not a valid message m was received in time, according to D3.25, a correct processor that receives m in communication phase j ($1 \leq j \leq T+1$) of its sub-BAP, checks if j is smaller than the number of processors in the path information of m. We assume, that every processor $i \in N$ stores the clock values (of its own processor clock) at which the communication phases of its sub-BAP end in an array $endcommphase_i$.

  Initially, the processors do not know when a BAP will start, and hence, any processor $i \in N$ does not know these clock values until i receives the first valid message from the BAP. Hence, for any processor $i \in N$, and any communication

34. Since clocks of faulty processors may run at arbitrary rates, we make no assumptions for the length or the presence of communication phases in *faulty* processors.

phase j, with $1 \le j \le T+1$, $endcommphase_i(j)$ is initialized to ∞.

As soon as processor i receives the first valid message from the BAP, say at clock value $C(i,t)$ [35] of its processor clock, i starts the first communication phase of its sub-BAP. Then, processor i is able to calculate at which clock values $C(i,t')$ of its processor clock each of the $T+1$ communication phases of its sub-BAP ends, since, by A3.9 (in Section 3.3.8.), i knows a priori the length L_k $(1 \le k \le T+1)$ of every communication phase of its sub-BAP. For all h, with $1 \le h \le T+1$, it holds that

$$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

By A3.9, for all k, with $1 \le k \le T+1$, L_k is the length of communication phase k, as viewed from i, where L_k is recursively defined by:

$$L_1 = (\tau_{max} - \tau_{min})\cdot(1+\rho)$$
$$L_2 = (\tau_{max} - \tau_{min})\cdot(1+\rho)^3 + (\Delta + \tau_{max} + \tau_{min})\cdot(1+\rho)$$
$$(\forall\, h \in [3,T+1]{::}\ L_h = [L_{h-1}\cdot(1+\rho)+\Delta]\cdot(1+\rho))$$

An authenticated self-synchronizing BAP satisfying A3.9. can be designed such that it satisfies the conditions IC1 and IC2 from Section 3.1.2. However, usually, an authenticated self-synchronizing BAP will be part of an authenticated self-synchronizing ICA. The lengths of the communication phases of an authenticated self-synchronizing ICA, viewed from a processor, are equal[36] or longer than the lengths of the communication phases of an authenticated self-synchronizing BAP, viewed from that processor. This implies, that, if the BAP is used as a part of an ICA, then the length of any communication phase in the authenticated self-synchronizing BAP should be as long as that of the corresponding communication phase in the authenticated self-synchronizing ICA. In other words, the lengths of the communication phases of the authenticated self-synchronizing BAP should be as indicated in A3.15. (instead of A3.9.). The proofs given in Section 3.3.9. are based on authenticated self-synchronizing BAPs which satisfy A3.9., however, it is straightforward to adapt the proofs such that they are based on authenticated self-synchronizing BAPs which satisfy A3.15. instead of A3.9. The verification of this is left to the reader.

- It is assumed that one processor (i.e., the source s) is triggered to start the BAP. This may be due to some external event (e.g., the receipt of a message belonging to an ICA which processor s did not yet participate in) or because s's processor clock reaches a certain predefined clock value. We will not further specify this event in the protocol given below.

35. $C(i,t)$ is the value of processor i's clock at time t. $C(i,t)$ is defined in assumption A3.2 in Section 3.3.8.

36. Mostly, in an ICA, every communication phase L_i (in A3.15.) will be **longer** than the same communication phase L_i (in A3.9) in a BAP. The length of a communication phase L_i $(1 \le i \le T+1)$ is only **equal** in the following cases. Communication phase L_1 is equal for a BAP (A3.9) and an ICA (A3.15.), only if $\tau_{max} = \tau_{min} = 0$. For $2 \le i \le T+1$, L_i is also equal for a BAP and an ICA, if $\rho=0$.

An authenticated self-synchronizing BAP can be described as follows:
Initially, for each processor $i \in N$, for all j, with $1 \le j \le T+1$, $endcommphase_i(j) = \infty$

For some processor $s \in N$:

Event: *Processor s is triggered to start the BAP at time C(s,t)*

Action: *Processor s acts as the source and generates and sends a message m(mv,(s;k)) to every processor $k \in N \setminus \{s\}$. The source accepts a copy of its own message in order to use it in the decision-making process. Furthermore, the source calculates at which clock values C(s,t') of its processor clock each of the T+1 communication phases of its sub-BAP ends. For all h, with $1 \le h \le T+1$, it holds that*

$$endcommphase_s(h) = C(s,t) + \sum_{k=1}^{h} L_k$$

For each processor $i \in N$:

Event: *$C(i,t) \ge endcommphase_i(T+1)$*

Action: *processor i decides on basis of the set of message values of the messages it has accepted during the multicast process of its sub-BAP. If this set of message values contains only one message value, then this message value is decided, otherwise an a priori agreed default value is taken as decision.*

Event: *message $mess_j(i)$ (with $1 \le j \le M_i$) received at time C(i,t)*

Action:

1. *Check if $mess_j(i)$ is a valid message, i.e., check if $mess_j(i) \in$ ValidMessages*
   If invalid($mess_j(i)$) then reject $mess_j(i)$ and abort, i.e., do no perform further actions on $mess_j(i)$.
   If valid($mess_j(i)$), then $mess_j(i)$ = m(mv,($\underline{\mathbf{q}}$)) for some path ($\underline{\mathbf{q}}$) $\in$ ValidPaths and some mv $\in$ MessageValues. If last(($\underline{\mathbf{q}}$)) $\neq$ i, then reject $mess_j(i)$ and abort, otherwise $mess_j(i)$ = m(mv,($\underline{\mathbf{p}}$;i)) for path ($\underline{\mathbf{p}}$) $\in$ ValidPaths with ($\underline{\mathbf{p}}$) = prefix(($\underline{\mathbf{q}}$)).
2. *Check if a valid message $mess_k(i)$ (with $1 \le k < j$) with path($mess_k(i)$) = ($\underline{\mathbf{p}}$;i) has already been accepted.*
   If such a message has already been accepted, then reject $mess_j(i)$ and abort.
3. *Check if m(mv,($\underline{\mathbf{p}}$;i)) has been received in time, i.e., check if C(i,t) $\le$ $endcommphase_i$(length(($\underline{\mathbf{p}}$;i))-1). If m(mv,($\underline{\mathbf{p}}$;i)) has been received in time, i accepts m(mv,($\underline{\mathbf{p}}$;i)), otherwise i rejects m(mv,($\underline{\mathbf{p}}$;i)).*
4. *If m(mv,($\underline{\mathbf{p}}$;i)) is the first valid message that i receives (and hence, accepts), i starts the first communication phase of its sub-BAP at receipt of m(mv,($\underline{\mathbf{p}}$;i)). Processor i now calculates at which clock values C(i,t') of its processor clock each of the T+1 communication phases of its sub-BAP ends. For all h, with $1 \le h \le T+1$, it holds that*

$$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

5. *If i accepted m(mv,($\underline{\mathbf{p}}$;i)) and path(m(mv,($\underline{\mathbf{p}}$;i))) contains fewer than*

*T+2 processors, for every processor h that is not in path(m(mv,(**p**;i))), i signs message m(mv,(**p**;i)) and i relays m(mv,(**p**;i)) to h, immediately after m(mv,(**p**;i)) was received by i. As stated in Section 3.3.3.3., relaying message m(mv,(**p**;i)) to any processor h results in m(mv,(**p**;i;h)) being sent to h.*

All correct processors are assumed to execute the above-described protocol. The behaviour of faulty processors, however, may deviate arbitrarily from the above-described protocol.

3.3.7. A description of an optimized authenticated self-synchronizing BAP

In the previous section, authenticated self-synchronizing BAPs have been described. Clearly, these BAPs are inefficient with regard to the number of messages communicated, since in these BAPs, every valid message *m*, that is received by a correct destination processor is relayed to all destination processors, the signature of which is absent in the path information of *m* (i.e., to all correct destination processors that did not yet receive *m* as well as to all faulty processors, the signature of which is absent in the path information of *m*).

In literature, several ***optimized BAPs*** have been designed, that are more efficient with regard to the number of messages exchanged (e.g., in [DoSt83, PoKr95]) if compared to the BAPs in [LaSP82]. In the optimized protocols, received messages are only relayed to a large enough subset of processors, instead of to all processors that did not yet receive the message (as, e.g., in [LaSP82]). However, the proposed optimized protocols in [DoSt83, PoKr95] assume a lock-step synchronous system and are based on the assumption that all correct processors start the protocol simultaneously.

In this section, we present an ***optimized authenticated self-synchronizing BAP*** (based on the BAPs in Section 3.3.6.), which works for any synchronous system, which does not require all processors to start the protocol simultaneously and which requires less communication overhead than the BAPs in the previous section. However, the reduction in the number of messages communicated in these optimized authenticated self-synchronizing BAPs is obtained at the cost of increased lengths of the $T+1$ communication phases of the sub-BAPs (for $T \geq 2$), if compared to the self-synchronizing BAPs described in Section 3.3.6. See Section 3.3.11. for details.

In an optimized authenticated self-synchronizing BAP, a correct processor relays every valid message it receives to a large enough subset of processors. We assume that these subsets are selected according to some predefined rules, known by all correct processors[37]. On basis of these so-called ***subset selection rules*** of a BAP, any correct processor expects that the path described in the path information of the first valid message it receives from the BAP, has a length greater than or equal to a certain k ($2 \leq k \leq T+1$). Faulty processors may relay received messages to arbitrary subsets of processors, different from those defined by the subset selection rules.

37. This is expressed in assumption A3.10 and A3.11 in Section 3.3.8. How these subset selection rules should be chosen is investigated below.

By relaying valid messages only to a subset of processors, protocol synchronization between all correct processors, as described in Section 3.3.1.1., may not be guaranteed, since the subset selection rules may be chosen such that some correct processors never receive a valid message. Therefore, we assume that the subset selection rules are chosen such that every valid message from the source is relayed to all correct processors within T relay steps[38]. In Section 3.3.10., we prove that under this assumption our BAPs guarantee Byzantine Agreement, i.e., satisfaction of the interactive consistency conditions IC1 and IC2.

The ***subset selection rules*** should be chosen such that the selected subsets satisfy the following properties.

We define $S_{(s)}$ as the subset of processors that should receive a message from the source s ($s \in N$). Furthermore, we define $S_{(\underline{\mathbf{p}}\,;\,i)}$ as the subset of processors to which processor i ($i \in N$) should relay any valid message $m(mv,(\underline{\mathbf{p}}\,;\,i))$ that it has accepted. We define $R_{(\underline{\mathbf{p}}\,;\,i)}$ as the set of processors that are present in the path described in the path information of $m(mv,(\underline{\mathbf{p}}\,;\,i))$, i.e.

$$R_{(\underline{\mathbf{p}}\,;\,i)} = \{a \in N \mid occurrences(a, (\underline{\mathbf{p}}\,;\,i)) = 1\}$$

Then the subsets should satisfy:

$$S_{(s)} = N \setminus \{s\} \qquad \text{if } T = 0$$
$$S_{(s)} \subset N \setminus \{s\} \wedge |\, S_{(s)} \,| \geq T{+}1 \qquad \text{if } T > 0$$

$$S_{(\underline{\mathbf{p}}\,;\,i)} = N \setminus R_{(\underline{\mathbf{p}}\,;\,i)} \qquad \text{if } length((\underline{\mathbf{p}}\,;\,i)) = T{+}1$$
$$S_{(\underline{\mathbf{p}}\,;\,i)} \subset N \setminus R_{(\underline{\mathbf{p}}\,;\,i)} \wedge |\, S_{(\underline{\mathbf{p}}\,;\,i)} \,| \geq min(T{+}1, |\, N \setminus R_{(\underline{\mathbf{p}}\,;\,i)} \,|\,)$$
$$\text{if } 2 \leq length((\underline{\mathbf{p}}\,;\,i)) \leq T{+}1$$

The remarks that were made in the previous section with regard to authenticated self-synchronizing BAPs are also applicable to optimized authenticated self-synchronizing BAPs, with that difference that, by A3.14 (in Section 3.3.8.), for all k, with $1 \leq k \leq T{+}1$, L_k is recursively defined by

$$L_1 = (\tau_{max} - \tau_{min})\cdot(1+\rho)$$
$$L_2 = (\tau_{max} - \tau_{min})\cdot(1+\rho)^3 + (\Delta + T\cdot\tau_{max}+\tau_{min})\cdot(1+\rho)$$
$$(\forall\, h \in [3,T{+}1]{::}\ L_h = [L_{h-1}\cdot(1+\rho)+\Delta]\cdot(1+\rho))$$

An optimized authenticated self-synchronizing BAP satisfying A3.14. can be designed such that it satisfies the interactive consistency conditions IC1 and IC2 given in Section 3.1.2. However, usually, an optimized authenticated self-synchronizing BAP will be part of an optimized authenticated self-synchronizing ICA. The lengths of the communication phases of an optimized authenticated self-synchronizing ICA, viewed from a processor, are equal[39] or longer than the lengths of the communication phases of an

38. This is expressed in assumption A3.12 in Section 3.3.8.
39. Mostly, in an ICA, every communication phase L_i (in A3.17.) will be **longer** than the same communication phase L_i (in A3.14.) in a BAP. The length of a communication phase L_i ($1 \leq i \leq T{+}1$) is only **equal** in the following cases. Communication phase L_1 is equal for a BAP (A3.14.) and an ICA (A3.17.), only if $\tau_{max} = \tau_{min} = 0$, or if $T = \tau_{min} = 0$. For $2 \leq i \leq T{+}1$, L_i is also equal for a BAP and an ICA, if $\rho{=}0$.

optimized authenticated self-synchronizing BAP, as viewed from that processor. This implies that, if the BAP is used as a part of an ICA, then the length of any communication phase in the optimized authenticated self-synchronizing BAP should be as long as that of the corresponding communication phase in the optimized authenticated self-synchronizing ICA that it is part of. In other words, if an optimized authenticated self-synchronizing BAP is used as part of an optimized authenticated self-synchronizing ICA, then the lengths of the communication phases of the optimized authenticated self-synchronizing BAP should be as indicated in A3.17. (instead of A3.14.). The proofs given in Section 3.3.10. are based on optimized authenticated self-synchronizing BAPs which satisfy A3.14., however, it is straightforward to adapt the proofs such that they are based on optimized authenticated self-synchronizing BAPs which satisfy A3.17. instead of A3.14. The verification of this is left to the reader.

An optimized authenticated self-synchronizing BAP can be described in a similar way as the authenticated self-synchronizing BAP given in the previous section:
Initially, for each processor $i \in N$*, for all* j*, with* $1 \leq j \leq T+1$, $endcommphase_i(j) = \infty$

For some processor $s \in N$*:*
Event: *Processor s is triggered to start the BAP at time* $C(s,t)$
Action: *Processor s acts as the source and generates and sends a message* $m(mv,(s;k))$ *to every processor* $k \in S_{(s)}$*. The source accepts a copy of its own message in order to use it in the decision-making process. Furthermore, the source calculates at which clock values* $C(s,t')$ *of its processor clock each of the T+1 communication phases of its sub-BAP ends. For all h, with* $1 \leq h \leq T+1$*, it holds that*

$$endcommphase_s(h) = C(s,t) + \sum_{k=1}^{h} L_k$$

For each processor $i \in N$*:*
Event: $C(i,t) \geq endcommphase_i(T+1)$
Action: *processor i decides on basis of the set of message values of the messages it has accepted during the multicast process of its sub-BAP. If this set of message values contains only one message value, then this message value is decided, otherwise, an a priori agreed default value is decided.*

Event: *message* $mess_j(i)$ *(with* $1 \leq j \leq M_i$*) received at time* $C(i,t)$
Action:
1. *Check if* $mess_j(i)$ *is a valid message, i.e., check if* $mess_j(i) \in$ *ValidMessages*
   If invalid($mess_j(i)$*) then reject* $mess_j(i)$ *and abort, i.e., do no perform further actions on* $mess_j(i)$*.*
   If valid($mess_j(i)$*), then* $mess_j(i) = m(mv,(\mathbf{q}))$ *for some path* $(\mathbf{q}) \in$ *ValidPaths and some* $mv \in$ *MessageValues. If* $last((\mathbf{q})) \neq i$*, then reject* $mess_j(i)$ *and abort, otherwise* $mess_j(i) = m(mv,(\mathbf{p};i))$ *for path* $(\mathbf{p}) \in$ *ValidPaths with* $(\mathbf{p}) = prefix((\mathbf{q}))$*.*
2. *Check if message* $m(mv,(\mathbf{p};i))$ *was expected according to i's subset selection rules, i.e., check if* $i \in S_{(\mathbf{p})}$*.*

If i $\notin S_{(\underline{\mathbf{p}})}$, then reject m(mv,($\underline{\mathbf{p}}$;i)) and abort.

3. *Check if a valid message mess$_k$(i) (with $1 \le k < j$) with path(mess$_k$(i)) = ($\underline{\mathbf{p}}$;i) has already been accepted.*
   If such a message has already been accepted, then reject mess$_j$(i) and abort.
4. *Check if m(mv,($\underline{\mathbf{p}}$;i)) has been received in time, i.e., check if C(i,t) $\le$ endcommphase$_i$(length(($\underline{\mathbf{p}}$;i))-1). If m(mv,($\underline{\mathbf{p}}$;i)) has been received in time, i accepts m(mv,($\underline{\mathbf{p}}$;i)), otherwise i rejects m(mv,($\underline{\mathbf{p}}$;i)).*
5. *If m(mv,($\underline{\mathbf{p}}$;i)) is the first valid message that i receives (and hence, accepts), i starts the first communication phase of its sub-BAP at receipt of m(mv,($\underline{\mathbf{p}}$;i)). Processor i now calculates at which clock values C(i,t') of its processor clock each of the T+1 communication phases of its sub-BAP ends. For all h, with $1 \le h \le T+1$, it holds that*

$$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

6. *If i accepted m(mv,($\underline{\mathbf{p}}$;i)) and path(m(mv,($\underline{\mathbf{p}}$;i))) contains fewer than T+2 processors, for every processor h $\in S_{(\underline{\mathbf{p}};i)}$, i signs message m(mv,($\underline{\mathbf{p}}$;i)) and i relays m(mv,($\underline{\mathbf{p}}$;i)) to h, immediately after m(mv,($\underline{\mathbf{p}}$;i)) was received by i. As stated in Section 3.3.3.3., relaying message m(mv,($\underline{\mathbf{p}}$;i)) to any processor h results in m(mv,($\underline{\mathbf{p}}$;i;h)) being sent to h.*

All correct processors are assumed to execute the above-described protocol. The behaviour of faulty processors, however, may deviate arbitrarily from the described protocol.

3.3.8. Assumptions for authenticated self-synchronizing BAPs

The assumptions we make for our authenticated self-synchronizing BAPs are:

A3.1. There exist fixed known upper and lower bounds (τ_{max} and τ_{min} respectively) on the time required to communicate a message from one correct processor to another processor in the system. Furthermore, $0 \le \tau_{min} \le \tau_{max}$.

A3.2. For any processor p and any real time t, let $C(p, t)$ be the value of p's clock at time t. Then, for any correct processor p in the system, and any two points t_1 and t_2 in real-time, we assume there exists a $\rho \ge 0$ (known by all correct processors), such that:

$$\frac{1}{1+\rho} \le \frac{C\left(p, t_1\right) - C\left(p, t_2\right)}{t_1 - t_2} \le (1+\rho)$$

Thus, we assume that the processor clocks of correct processors are ρ-bounded. Notice that the value $C(p,t)$ of the clock of a correct processor p may deviate arbitrarily from real-time t (Thus, arbitrary clock skew between processor clocks of the processors in the system is allowed).

A3.3. We assume the existence of a nonnegative real-time bound Δ on clock value measurement uncertainty, known by all correct processors. The first time any correct processor p detects that its processor clock indicates a clock value greater than or equal to a certain value v, it is assumed that p's clock indicated v at most Δ ago.

A3.4. A processor may concurrently execute different BAPs. We assume that messages of different BAPs can be distinguished, such that every correct processor can determine when it receives the first valid message of a new BAP.

A3.5. A perfect communication link is available between every pair of correct processors in the system. This assumption is justified by the fact that we can model a link failure as a failure of one of its adjacent processors [SiLL90].

A3.6. Every valid message m contains a message value and authenticated path information. Knowledge of the path information of a valid message m is sufficient to group m in the decision-making process and to determine a set of processors which includes all correct processors that did not yet receive m. (See Section 3.3.1.2.)

For authenticated self-synchronizing BAPs, we make the following additional assumptions:

A3.7. Every valid message m that is timely received by a correct processor is relayed to all processors that are not in the path information of m. This implies that m is sent to all correct processors that did not yet receive m. (See Section 3.3.1.2.)

A3.8. If the source is correct, in every destination processor d, a valid message from the source arrives in d within a real-time interval of length between τ_{min} and τ_{max} after the source started the BAP.

A3.9. Every processor in the system may have a different view of time and of the start and end of communication phases in a BAP. For all i, with $1 \le i \le T+1$, for any correct processor p, let L_i be the length of communication phase i, as viewed from p. Then[40]:

$$L_1 = (\tau_{max} - \tau_{min})\cdot(1+\rho)$$

$$L_2 = (\tau_{max} - \tau_{min})\cdot(1+\rho)^3 + (\Delta + \tau_{max} + \tau_{min})\cdot(1+\rho).$$

$$(\forall\, i \in [3, T+1] :: L_i = [L_{i-1} \cdot (1+\rho) + \Delta]\cdot(1+\rho)).$$ ❑

As already stated in Section 3.3.6., if the authenticated self-synchronizing BAP is part of an authenticated self-synchronizing ICA, then we assume that the lengths of the various communication phases in the BAP are as indicated in A3.15. instead of A3.9.

For optimized authenticated self-synchronizing BAPs, instead of assumption A3.7 through A3.9, we assume:

A3.10. Every valid message that is timely received by a correct processor is relayed to a large enough subset of processors according to the subset selection rules for that message.

A3.11. Every correct processor knows the subset selection rules of the BAP.

A3.12. The subset selection rules are such that every valid message from the source is relayed to all correct processors within T relay steps.

A3.13. If the source is correct, in every destination processor d selected by the subset selection rules of the source, a valid message from the source arrives in d

40. In Section 3.3.9., it will be proved that with these values for L_i (with $1 \le i \le T+1$), Byzantine Agreement can be guaranteed.

within a real-time interval of length between τ_{min} and τ_{max} after the source started the BAP.

A3.14. Every processor in the system may have a different view of time and of the start and end of communication phases in a BAP. For all i, with $1 \le i \le T+1$, for any correct processor p, let L_i be the length of communication phase i, as viewed from p. Then[41]:

$$L_1 = (\tau_{max} - \tau_{min})\cdot(1+\rho)$$

$$L_2 = (\tau_{max} - \tau_{min})\cdot(1+\rho)^3 + (\Delta + T\cdot\tau_{max} + \tau_{min})\cdot(1+\rho).$$

$$(\forall\, i \in [3, T+1] :: L_i = [L_{i-1}\cdot(1+\rho) + \Delta]\cdot(1+\rho)).$$ ❑

As already stated in Section 3.3.7., if the optimized authenticated self-synchronizing BAP is part of an optimized authenticated self-synchronizing ICA, then we assume that the lengths of the various communication phases in the BAP are as indicated in A3.17. instead of A3.14.

Notice that no assumptions are made about the behaviour of faulty processors. They may send arbitrary messages at arbitrary times, or refuse to relay received messages. Their processor clocks may run at arbitrary rates. Furthermore, faulty processors may collude, i.e., they may use each other’s signature to sign messages. However, we assume that faulty processors cannot prevent correct processors from meeting the assumptions stated in this section. In particular, we assume guaranteed communication from a correct processor to other processors in the system (i.e., satisfaction of assumption A3.1). In practice, there exists the possibility of faulty processors continuously generating new messages, as a result of which network congestion may occur. This problem is not considered here, because we think that no deterministic fault-tolerant solution to this problem exists.

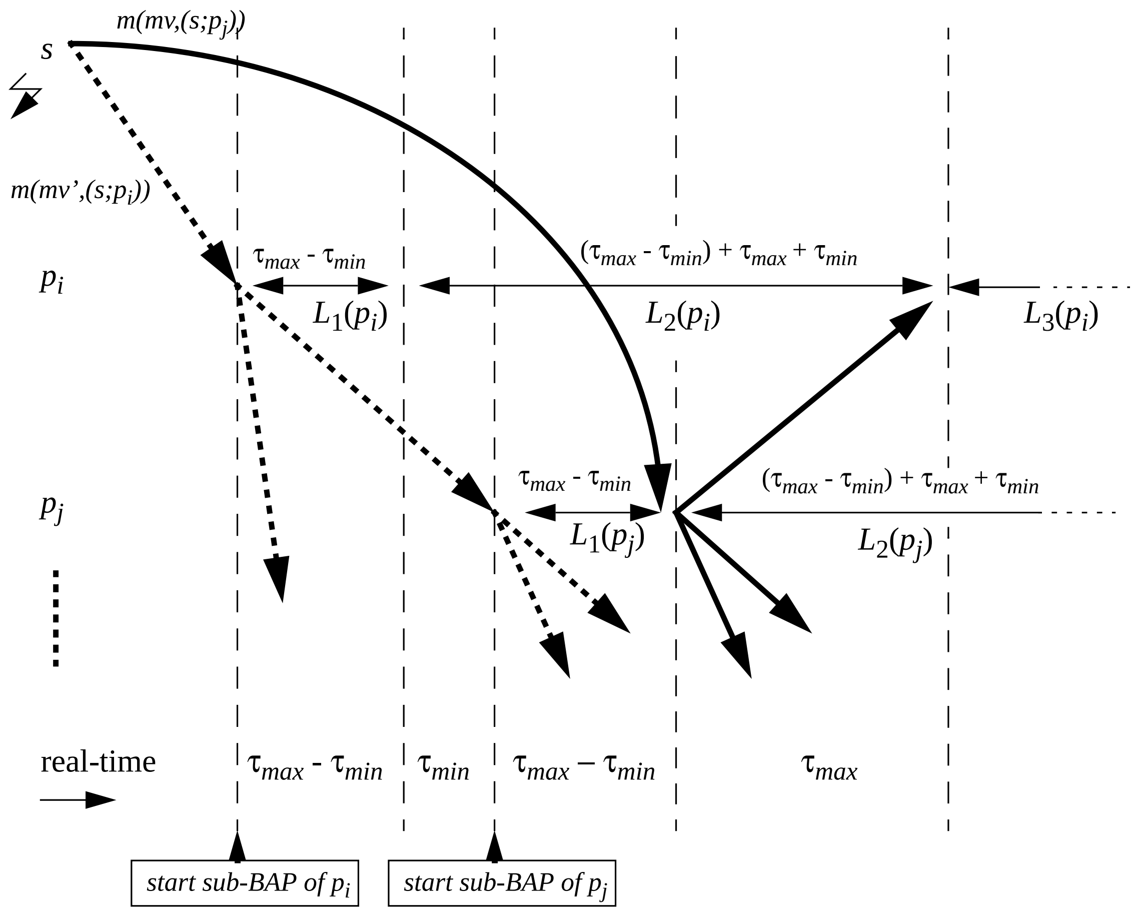
In Section 3.3.9. respectively 3.3.10., we prove that the BAPs presented in Section 3.3.6. respectively 3.3.7. satisfy the interactive consistency conditions IC1 and IC2 in a synchronous system of N processors, up to T of which may behave maliciously, provided that the above-stated assumptions hold.

3.3.9. Proofs for authenticated self-synchronizing BAPs

Assume a deterministic authenticated self-synchronizing BAP is executed in a fully-connected synchronous system $\mathcal{N}$ consisting of N processors (including a source s), up to T of which may behave maliciously. Assume that assumptions A3.1 through A3.9 and the unforgeability properties UP1 and UP2 hold. We now prove that, in $\mathcal{N}$, the interactive consistency conditions IC1 and IC2 are implied by assumptions A3.1 through A3.9 and the unforgeability properties UP1 and UP2.

In an example in Figure 3.8. through 3.10., in order of increasing complexity, we indicate how protocol synchronization between correct processors p_i and p_j (with p_i and p_j $\in \mathcal{N}$) is achieved in an authenticated self-synchronizing BAP, in the presence of a faulty source s ($s \in \mathcal{N}$). Proofs for the general case can be found in the lemma’s and theorems in this section. Figure 3.10. sketches the general case for arbitrary nonnega-

41. In Section 3.3.10., it will be proved that with these values for L_i (with $1 \le i \le T+1$), Byzantine Agreement can be guaranteed.



$\rho = 0 \wedge \Delta = 0$

$(\forall\ q \in\ N : \forall\ k \in\ [1,T+1]\text{:: } correct(q) \Rightarrow L_k(q) = L_k)$

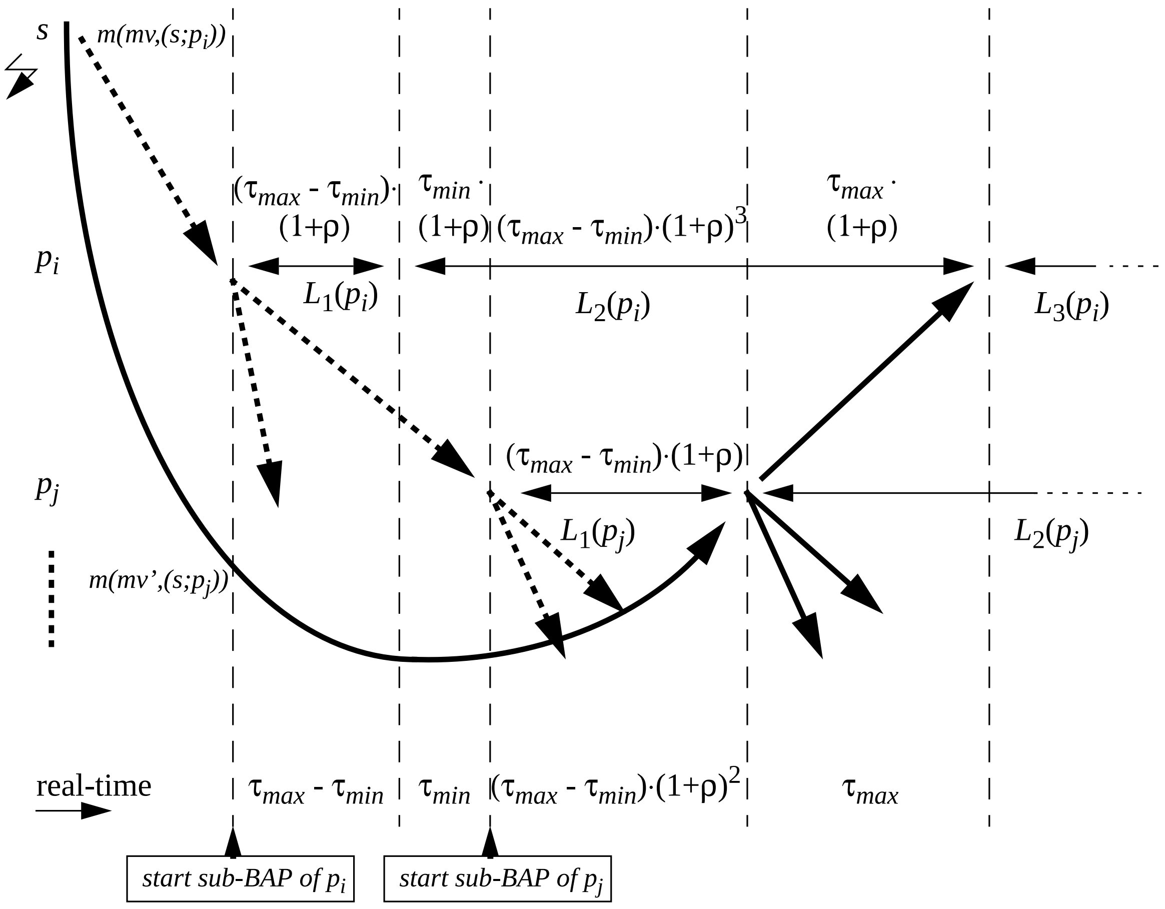
$L_1 = \tau_{max} - \tau_{min}$

$(\forall\ k \in\ [2,T+1] :: L_k = 2\tau_{max})$

For processor p_i and p_j it holds that

$(\forall\ k \in\ [1,T+1] :: L_k(p_i) = L_k(p_j) = L_k)$

Figure 3.8. *Protocol synchronization between correct processors p_i and p_j in an authenticated self-synchronizing BAP, in case the source s is faulty and $\rho = 0$ and $\Delta = 0$.*



$\rho \geq 0 \wedge \Delta = 0$

$(\forall\, p \in N : \forall\, k \in [1,T+1] :: correct(p) \Rightarrow (1 / (1+\rho))\cdot L_k(p) \leq L_k \leq (1+\rho)\cdot L_k(p))$

$L_1 = (\tau_{max} - \tau_{min})\cdot(1+\rho)$

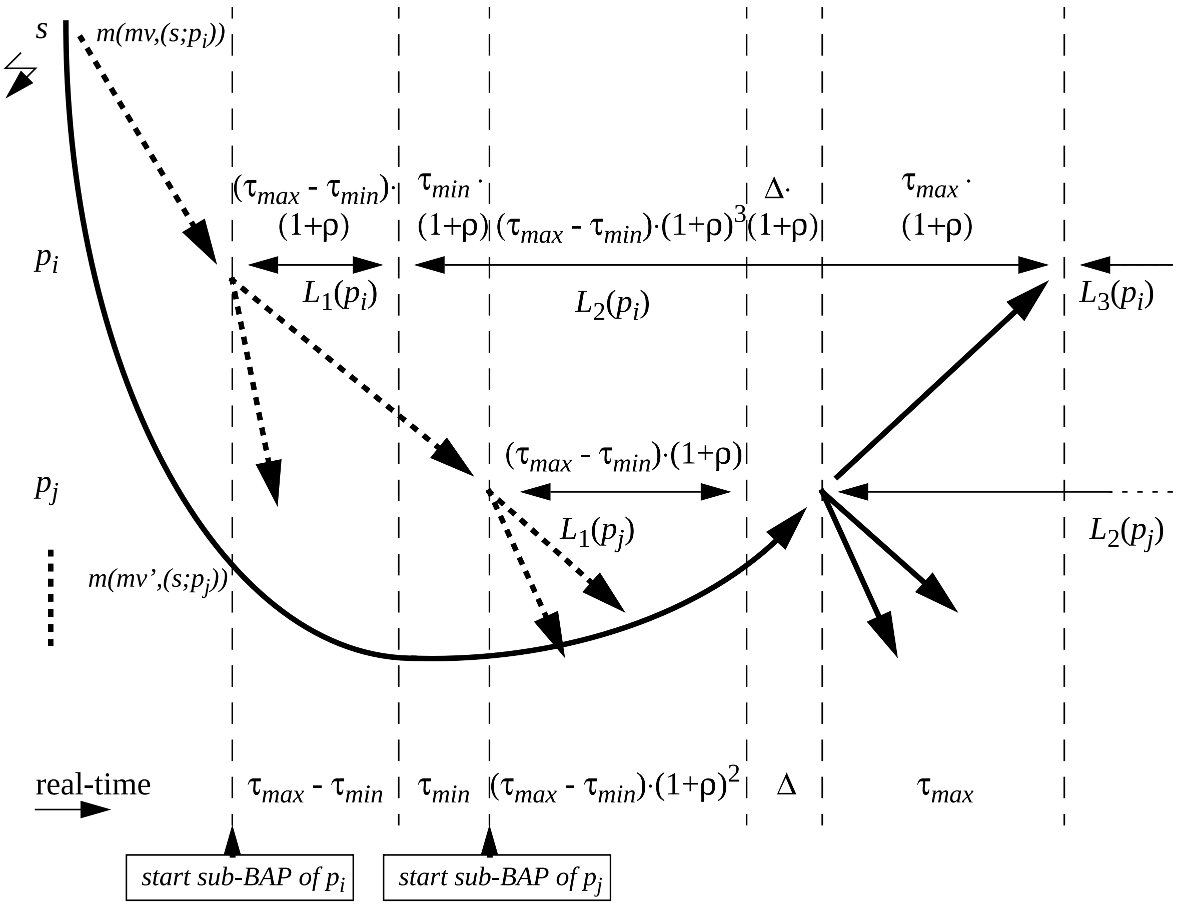
$L_2 = (\tau_{max} - \tau_{min})\cdot(1+\rho)^3 + (\tau_{max} + \tau_{min})\cdot(1+\rho)$

$(\forall\, k \in [3,T+1] :: L_k = L_{k-1}\cdot(1+\rho)^2)$

For processor p_i and p_j it holds that

$(\forall\, k \in [1,T+1] :: (1/(1+\rho))\cdot L_k(p_i) = (1+\rho)\cdot L_k(p_j) = L_k)$

Figure 3.9. *Protocol synchronization between correct processors p_i and p_j in an authenticated self-synchronizing BAP, in case the source s is faulty and $\rho \geq 0$ and $\Delta = 0$.*



$\rho \geq 0 \wedge \Delta \geq 0$

$(\forall\, p \in N : \forall\, k \in [1,T+1] :: correct(p) \Rightarrow (1 / (1+\rho))\cdot L_k(p) \leq L_k \leq (1+\rho)\cdot L_k(p))$

$L_1 = (\tau_{max} - \tau_{min})\cdot(1+\rho)$

$L_2 = (\tau_{max} - \tau_{min})\cdot(1+\rho)^3 + (\Delta + \tau_{max} + \tau_{min})\cdot(1+\rho)$

$(\forall\, k \in [3,T+1] :: L_k = [L_{k-1}\cdot(1+\rho) + \Delta]\cdot(1+\rho))$

For processor p_i and p_j it holds that

$(\forall\, k \in [1,T+1] :: (1/(1+\rho))\cdot L_k(p_i) = (1+\rho)\cdot L_k(p_j) = L_k)$

Figure 3.10. *Protocol synchronization between correct processors p_i and p_j in an authenticated self-synchronizing BAP, in case the source s is faulty and $\rho \geq 0$ and $\Delta \geq 0$.*

tive ρ and Δ. To make it easier to understand Figure 3.10., in Figure 3.8. and 3.9., respectively, we have depicted the protocol synchronization in simpler cases ($\rho = 0 \wedge \Delta = 0$, and $\rho \geq 0 \wedge \Delta = 0$, respectively). In the example, the time needed to communicate messages, the speeds at which the processor clocks of different processors operate, and the clock value measurement uncertainties have been chosen such, that the difference in protocol synchronization between p_i and p_j is maximal. The length of the communication phases $L_k(p_i)$ (respectively $L_k(p_j)$) (with $2 \leq k \leq T+1$) of the BAP, as viewed by processor p_i (respectively p_j) is determined by the maximal difference in protocol synchronization, and the requirement that each message that is timely received (and hence, may have been sent in time) should be accepted. The length of the first communication phase $L_1(p_i)$ (respectively $L_1(p_j)$) is determined by the requirement that every valid message that could have been sent in time by the source s, should be accepted. The real-time lengths of several time intervals have been indicated at the bottom of the figures. Since the clocks of correct processors may have a bounded drift from real-time, the length of a time interval, measured by the clock of a correct processor may be up to a factor $(1+\rho)$ longer or shorter than the real-time length of this time interval. The lengths of these time intervals, as measured by the processor clocks of p_i and p_j are indicated above the communication phases $L_k(p_i)$ and $L_k(p_j)$ (with $1 \leq k \leq T+1$), respectively.

LEMMA 3.5.1.
If a correct processor c receives the first valid message m of a BAP with path information describing a path containing n processors ($2 \leq n \leq T+1$), c knows that the BAP has been going on for at least τ_{min}.

Proof:
Follows directly from assumption A3.1. ❑

Notice that, although it is very well possible that the BAP is going on for longer than τ_{min}, c may not be able to conclude that[42]. See Figure 3.11. For this reason, at receipt of the first valid message of a BAP, any correct processor starts its sub-BAP in the ***first*** communication phase. In this way, as is proved in Theorem 3.5., 3.6., and 3.7., it is guaranteed that this processor accepts all valid messages that ***may be*** in time[43]. Otherwise, a message could be rejected although it was indeed sent in time, and this might lead to violation of interactive consistency condition IC2.

THEOREM 3.5.
If the source which initiates the BAP is *correct*, any valid message sent directly from

42. Notice that for any n ($2 \leq n \leq T+1$) there always exists a situation in which the BAP has only been going on for τ_{min} at the moment that c receives a valid message with path information that describes a path containing n processors.
For n=2, this can easily be seen. In this case, message m may be sent directly from the source to c. Any valid message from a correct source may arrive τ_{min} after the source started the BAP (by A3.8).
For $3 \leq n \leq T+1$, this lower bound τ_{min} is reached, if there is collusion between faulty processors. Then, a faulty processor may sign m with the signatures of n faulty processors (starting with the signature of the source), and send it to c within a minimum real-time interval of length τ_{min}.

the source to a correct processor *c* is timely received and accepted in processor *c*.

Proof:
Assume that the BAP is initiated by a correct source *s*. Then, the source *s* initiates the BAP by sending a valid message *m*(*mv*,(*s;c*)) to any processor *c* in the system. Since the source s is assumed to be correct, assumption A1 implies that any valid message *m*(*mv*,(*s;c*)) sent by the source *s* to any processor *c* in the system will arrive in processor *c* after a real-time interval of at most τ_{max} after the source sent it. We will prove that, provided that processor *c* is correct, message *m*(*mv*,(*s;c*)) is timely received and accepted by processor *c*. We distinguish between the case that *m*(*mv*,(*s;c*)) is the first valid message from the BAP that processor *c* receives, and the case that *m*(*mv*,(*s;c*)) is not the first valid message from the BAP that processor *c* receives, and we show that in both cases, message *m*(*mv*,(*s;c*)) is timely received and accepted in processor *c*.

Case 1: *m*(*mv*,(*s;c*)) is the first valid message from the BAP that processor *c* receives.
In this case, processor *c* will start the first communication phase of its sub-BAP at receipt of *m*(*mv*,(*s;c*)). From the description of the protocol, we know that any valid message received in any correct processor *d* in or before communication phase *i* ($1 \leq i \leq T+1$) is timely received and accepted in processor *d*, provided that the length of the path described in the path information of the message is greater than *i*. Hence, message *m*(*mv*,(*s;c*)) is timely received and accepted by processor *c*, since it is received before communication phase 1 of *c*'s sub-BAP, and the length of path (*s;c*) is equal to 2.

43. Notice that it is not necessary to require that all valid messages that are accepted were ***indeed*** sent in time. To make this clear, we consider two BAPs, B1 and B2, which both satisfy assumptions A3.1 through A3.9 and are identical except that, in B1, all valid messages that are not sent in time are rejected, whereas in B2 there may be some correct processors that accept one or more valid messages that are not sent in time (but thought to be sent in time by those processors). We show that, if the interactive consistency conditions are satisfied in B1, they are also satisfied in B2. Assume that the interactive consistency conditions are satisfied in B1. We now show that they are also satisfied in B2.

Assume the source is ***correct***. Then, in B1, the decision of all correct processors is equal to the message value that the source sent (by IC1). All valid messages will contain the same message value, and carry the signature of the source. Since the decision taken in a correct processor is based on *valid* messages only, acceptance of a valid message which is not sent in time in a certain correct processor in B2 does not result in a decision different from that taken in that processor in B1. Thus, the decisions taken by the correct processors are equal in B1 and B2, and thus, in this case, the interactive consistency conditions are also satisfied in B2.

Now assume that the source is ***faulty***. Notice that IC2 is trivially satisfied. In this case, the source may create valid messages with differing message values. Then, if, in B2, a correct processor *c* accepts a valid message *m* that is not sent in time (this may happen if *c* thinks that *m* may be sent in time), this may result in a decision in B2 different from the decision that *c* would have taken in B1. However, in B2, by Theorem 3.7., all correct processors will accept *m* and include it in their decision-making process. Every correct processor in B2 also accepts all message values it accepted in B1. By Theorem 3.7., in B2, all correct processors accept the same set of valid messages on which they base their decision, and thus, IC1 is satisfied.

Thus, in both cases, whether or not the interactive consistency conditions are satisfied is not influenced by the acceptance of valid messages that are not sent in time (but thought to be sent in time) by any correct processor.

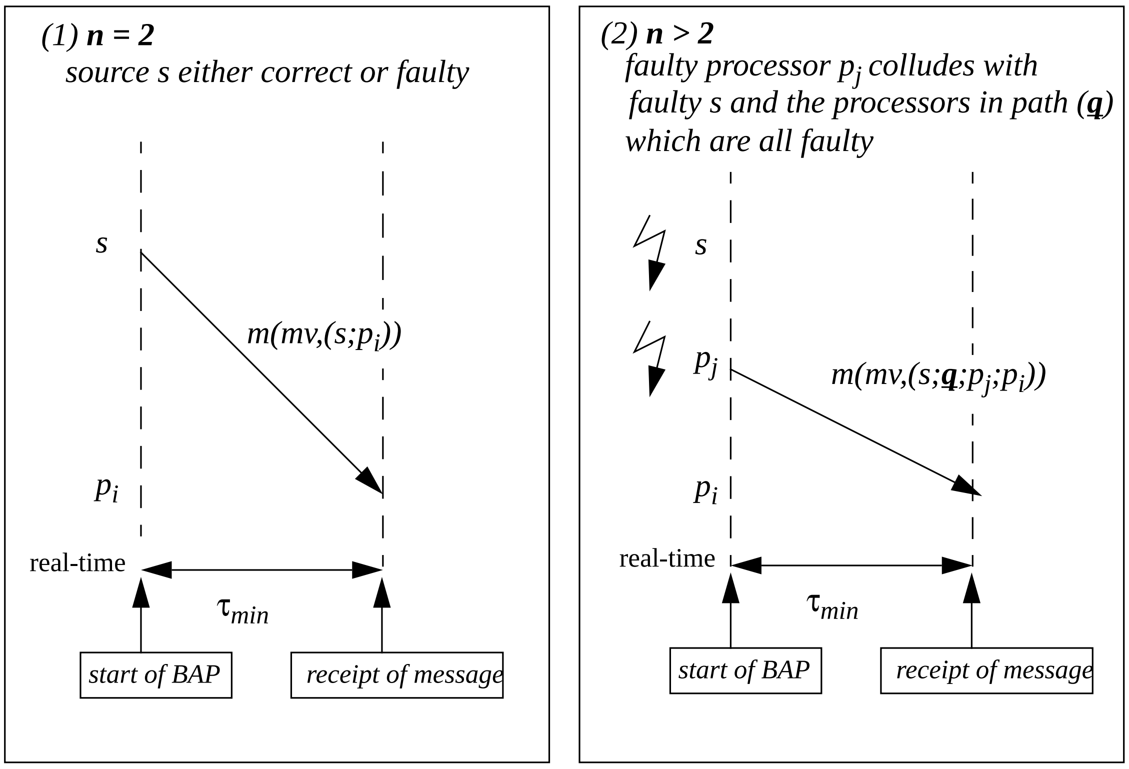


Figure 3.11. *The moment at which a correct processor p_i receives the first valid message of a BAP, with path information describing a path containing n processors (n > 1), p_i knows that the BAP has been going on for at least τ_{min}.*

Case 2: *m*(*mv*,(*s;c*) is not the first valid message from the BAP that processor *c* receives.

In this case, processor *c* must have received the first valid message *m*(*mv*,(*s;**p**;c*)) (for some non-empty path (***p***)) from the BAP before it receives *m*(*mv*,(*s;c*)). Since we assumed that the source *s* is correct, from unforgeability property UP2, we conclude that message *m*(*mv*,(*s;**p**;c*)) must be the result of relaying a valid message *m*(*mv*,(*s;x*)) (with *x* = *first*((***p***))), which has been created by the source *s*. The source *s* has created this message *m*(*mv*,(*s;x*)) at the moment at which it has initiated the BAP. Message *m*(*mv*,(*s;**p**;c*)) which was received by processor *c* can only be the result of signing message *m*(*mv*,(*s;x*)) received by processor *x* with the signatures of all processors in path (***p***).

The earliest moment at which message *m*(*mv*,(*s;**p**;c*)) can arrive in processor *c*, is a real -time interval with length of at least $2\tau_{min}$ after the source *s* has initiated the BAP, i.e., in case processor *x* knows the signatures of all processors in path (***p***) (either because path (***p***) only contains the identification of processor *x*, or if all processors in path (***p***) are faulty and processor *x* colludes with all other processors, the identification of which is present in path (***p***)). This situation may occur, if processor *x* receives *m*(*mv*,(*s;c*)) from the source, τ_{min} after the source has initiated the BAP (which is the minimum time needed to communicate *m*(*mv*,(*s;c*)) from *s* to *c*), and *x* creates *m*(*mv*,(*s;**p**;c*)) from *m*(*mv*,(*s;c*)) and sends it in a time interval of length τ_{min} to processor *c*.

At receipt of $m(mv,(s;\underline{\mathbf{p}};c))$, processor c will start the first communication phase of its sub-BAP. The first communication phase of c's sub-BAP has a length of $(\tau_{max}-\tau_{min})\cdot(1+\rho)$, as measured by c's processor clock. Since c's processor clock may run up to a factor $(1+\rho)$ faster than real-time, the first communication phase of c's sub-BAP may take up to a factor $(1+\rho)$ shorter to complete. I.e., the minimum real-time length of the first communication phase of c's sub-BAP is equal to $(\tau_{max} - \tau_{min})$. Since, furthermore, processor c starts the first communication phase of its sub-BAP at or after $2\tau_{min}$ after the source initiated the BAP, the earliest time at which processor c may conclude the first communication phase of its sub-BAP is after a real-time interval of at least $2\tau_{min} + (\tau_{max} - \tau_{min}) = (\tau_{max} + \tau_{min})$ after the source has initiated the BAP.

Since the source s is correct, it will send a valid message $m(mv,(s;c))$ to processor c, at the moment it initiates the BAP. This message will arrive in c after a real-time interval with length of at most τ_{max} after the source initiated the BAP. Hence, this message, with path information describing a path of length 2, always arrives in c in or before communication phase 1 of c's sub-BAP. Thus, message $m(mv,(s;c))$ is always timely received and accepted by processor c.

This proves Theorem 3.5. ❑

LEMMA 3.6.1.
Once a correct processor p has concluded the first communication phase of its sub-BAP, then, within a real-time interval of $(\tau_{max} - \tau_{min})\cdot(1+\rho)^2 + \tau_{min} + \Delta$, all other correct processors have also concluded the first communication phase of their sub-BAP.

Proof:
Notice that the maximum real-time difference between the end of the first communication phase of the sub-BAPs of two different correct processors p and q is determined by the maximum real-time difference between the start of the first communication phase of the sub-BAP of processor p and q, and by the maximum real-time difference in the length of the first communication phase in the sub-BAPs of processor p respectively, of processor q.

The real-time length of any communication phase i $(1 \leq i \leq T+1)$ of the sub-BAP of any correct processor c is ***maximal***, if c's processor clock runs a factor $(1+\rho)$ slower than real-time, and c detects the end of communication phase i only after a real-time interval of length Δ after its processor clock indicates that communication phase i is over. By A3.9, for any correct processor c, the length L_1 of the first communication phase of c's sub-BAP, as measured by c's processor clock, is equal to $(\tau_{max} - \tau_{min})\cdot(1+\rho)$. Hence, the *maximum* real-time length of the first communication phase of the sub-BAP of any correct processor c is $(1+\rho)\cdot L_1 + \Delta$, which is equal to $(\tau_{max} - \tau_{min})\cdot(1+\rho)^2 + \Delta$.

The real-time length of any communication phase i $(1 \leq i \leq T+1)$ of the sub-BAP of any correct processor c is ***minimal***, if c's processor clock runs a factor $(1+\rho)$ faster than real-time, and c immediately detects the end of communication phase i, as soon as its processor clock indicates that communication phase i is over. Hence, the *minimum* real-time length of the first communication phase of the sub-BAP of any correct processor c is L_1 / (1+ρ), which is equal to (τ_{max} - τ_{min}).

The real-time difference between the start of the first communication phase of the sub-BAP of two different correct processors is ***maximal***, if one of the correct processors, say p, is the first correct processor[44] that starts the first communication phase of its sub-BAP, and the other correct processor, say q, only starts the first communication phase of its sub-BAP at receipt of the first valid message from processor p, and the maximum real-time interval τ_{max} is needed to communicate this message from p to q. Hence, the maximum real-time difference between the start of the first communication phase of the sub-BAP of two correct processors is equal to τ_{max}. This can be seen as follows.

The above situation may occur, if processor p is the source which initiates the BAP, or if the BAP is initiated by a set of colluding faulty processors, and processor p is the first correct processor that receives a valid message from the BAP.

If processor p is the source which initiates the BAP, processor p will broadcast a message to all other processors in the system, and start the first communication phase of its sub-BAP. The message sent from processor p to processor q will arrive after a real-time interval of τ_{max}, and cause processor q to start the first communication phase of its sub-BAP at receipt of this message, if this message is the first valid message that q receives, i.e., if processor q has not previously received a valid message from another processor r, which has relayed to q the message received from p.

If processor p is the first correct processor that receives a valid message from a BAP initiated by a set of colluding faulty processors, then at receipt of this message, processor p will start the first communication phase of its sub-BAP, and relay the received message m to all processors, the identification of which is not in the path described by the path information of m. The message is also relayed to processor q, since the identification of processor q cannot be present in the path described in the path information of m, since unforgeability property UP1 in Section 3.3.3. implies that, in this case, q must have received and accepted m, and hence, q must have received a valid message from the BAP before p receives it. This contradicts our assumption that processor p was the first correct processor that receives a valid message from the BAP.

The message sent from processor p to processor q will arrive after a real-time interval of τ_{max}, and cause processor q to start the first communication phase of its sub-BAP at receipt of this message, if this message is the first valid message that q receives, i.e., if processor q has not previously received a valid message from another processor r.

The real-time difference between the end of the first communication phase of the sub-

44. Notice that ***more than one*** correct processor may be the first to start the first communication phase of its sub-BAP. Viz., the BAP may be initiated by a set of colluding faulty processors, and they may cause different valid messages to simultaneously arrive in more than one correct processor. Each of these correct processors will be triggered to start the first communication phase of its sub-BAP at receipt of this message, so each of these correct processors is the first to start the first communication phase of its sub-BAP.

BAPs of two different correct processors is ***maximal***, if the real-time difference between the start of the first communication phase is maximal, and the real-time difference in lengths of the first communication phase of the sub-BAPs in the different processors is maximal, i.e., if the length of the first communication phase of the sub-BAP of the correct processor that starts its sub-BAP first is minimal, whereas the length of the first communication phase of the sub-BAP of the correct processor that starts its sub-BAP τ_{max} later is maximal.

Let p be the first correct processor that starts its sub-BAP, say at time $t_{earliest\_start}$. Then, the earliest time at which processor p can conclude the first communication phase of its sub-BAP is at time $t_{earliest\_end} = t_{earliest\_start} + (\tau_{max} - \tau_{min})$ (i.e., if the real-time length of the first communication phase of p's sub-BAP is minimal). Then, the latest moment at which any other correct processor q can start the first communication phase of its sub-BAP, is at real-time $t_{latest\_start} = t_{earliest\_start} + \tau_{max}$ (i.e., in case q starts its sub-BAP as a result of receiving the first valid message of the BAP from p, τ_{max} after p sent it). The latest moment at which q can conclude the first communication phase of its sub-BAP is at $t_{latest\_end} = t_{latest\_start} + (\tau_{max} - \tau_{min})\cdot(1+\rho)^2 + \Delta$ (i.e., in case the real-time length of the first communication phase of q's sub-BAP is maximal).

Hence, as soon as a correct processor has concluded the first communication phase of its sub-BAP, after a real-time interval of at most

$$t_{latest\_end} - t_{earliest\_end} =$$
$$(t_{earliest\_start} + \tau_{max} + (\tau_{max} - \tau_{min})\cdot(1+\rho)^2 + \Delta) - (t_{earliest\_start} + (\tau_{max} - \tau_{min})) =$$
$$(\tau_{max} - \tau_{min})\cdot(1+\rho)^2 + \tau_{min} + \Delta,$$

all other correct processors have also concluded the first communication phase of their sub-BAP. This concludes Lemma 3.6.1. ❑

LEMMA 3.6.2.
After communication phase i ($1 \le i \le T$), in real-time, the sub-BAPs of all correct processors are synchronized within a real-time bound of $(L_{i+1} / (1+\rho)) - \tau_{max}$.

Proof:
We will prove this lemma by induction on i.

Basis: *i* = 1.
From Lemma 3.6.1. we know that after communication phase 1, the sub-BAPs of all correct processors have been synchronized within a real-time bound of $(\tau_{max} - \tau_{min})\cdot(1+\rho)^2 + \tau_{min} + \Delta$, which is equal to $(L_2 / (1+\rho)) - \tau_{max}$.

Induction step: *i* > 1.
We assume that Lemma 3.6.2. holds for $j = i - 1$ ($2 \le i \le T$). Thus, the induction hypothesis is:
After communication phase $i - 1$($2 \le i \le T$), in real-time, the sub-BAPs of all correct processors are synchronized within a real-time bound of $L_i / (1+\rho) - \tau_{max}$.

Now, we prove that Lemma 3.6.2. also holds for $j = i$.

Notice that the maximal real-time difference in synchronization occurs between the sub-BAP of a correct processor *s* which runs a factor (1+ρ) slower than real-time and that of a correct processor *f* which runs a factor (1+ρ) faster than real-time. The real-time difference is maximal if processor *f* immediately detects the end of its communication phase, whereas processor *s* only detects it after real-time Δ.

For all *i* (with $1 \leq i \leq T+1$) and any correct processor *p,* let $L_i(p)$ be the real-time length of communication phase *i* in *p*. For a correct processor *s* that runs a factor (1+ρ) slower than real-time, the maximum real-time length of communication phase *i* in *s* is $L_i(s) = L_i \cdot (1+\rho) + \Delta$. For a correct processor *f* that runs a factor (1+ρ) faster than real-time, the minimum real-time length of communication phase *i* in *f* is $L_i(f) = L_i / (1+\rho)$. So, the real-time difference in protocol synchronization increases by an amount less than or equal to $L_i(s) - L_i(f)$. From the induction hypothesis, we know that the real-time difference after communication phase $i - 1$ is less than or equal to $L_i / (1+\rho) - \tau_{max}$, which is equal to $L_i(f) - \tau_{max}$. After communication phase *i*, the maximal real-time difference in protocol synchronization is $(L_i(f)-\tau_{max})+(L_i(s)-L_i(f)) = L_i(s) - \tau_{max}$, which is equal to $L_i \cdot (1+\rho) + \Delta - \tau_{max}$. Since, by A3.9, for $i > 1$, $L_{i+1} = [L_i \cdot (1+\rho) + \Delta]\cdot (1+\rho)$, the maximal real-time difference in protocol synchronization after communication phase *i* is $L_i \cdot (1+\rho) + \Delta - \tau_{max} = L_{i+1} / (1+\rho) - \tau_{max}$. This proves Lemma 3.6.2. ❑

LEMMA 3.6.3.
Let $m(mv,(\underline{\boldsymbol{p}}))$ be any valid message which is timely received and accepted by a correct processor *c* in communication phase *i* ($1 \leq i \leq T$) of *c*'s sub-BAP and assume that *c* relays $m(mv,(\underline{\boldsymbol{p}}))$ to a correct processor *d*, i.e., processor *c* sends $m(mv,(\underline{\boldsymbol{p}};d))$ to *d*. Then, message $m(mv,(\underline{\boldsymbol{p}};d))$ arrives in processor *d* in or before communication phase *i*+1 of *d*'s sub-BAP.

Proof:
Assume we have two correct processors *c* and *d*. Assume furthermore that *c* receives in communication phase *i* of its sub-BAP ($1 \leq i \leq T$) a valid message $m(mv,(\underline{\boldsymbol{p}}))$, say at a certain time *t*. Assume that the identification of *d* is not present in path ($\underline{\boldsymbol{p}}$), i.e., the path described by the path information of $m(mv,(\underline{\boldsymbol{p}}))$. Then, immediately after having received $m(mv,(\underline{\boldsymbol{p}}))$, at time *t*, *c* relays $m(mv,(\underline{\boldsymbol{p}}))$ to *d*, i.e., *c* sends a message $m(mv,(\underline{\boldsymbol{p}};d))$ to *d*. Now, we must prove that $m(mv,(\underline{\boldsymbol{p}};d))$ arrives in *d* in or before communication phase *i*+1 of *d*'s sub-BAP.

Since *c* sends $m(mv,(\underline{\boldsymbol{p}};d))$ at time *t* to *d*, and *c* is a correct processor, $m(mv,(\underline{\boldsymbol{p}};d))$ arrives in *d* at or before $t + \tau_{max}$. Hence, it remains to prove that *d* concludes communication phase *i*+1 of its sub-BAP at or after $t + \tau_{max}$.

According to Lemma 3.6.2., at the end of communication phase *i* ($1 \leq i \leq T$), the sub-BAPs of all correct processors are synchronized within a real-time bound *B*, with $B = L_{i+1} / (1+\rho) - \tau_{max}$. Since the maximal real-time difference in protocol synchronization between correct processors increases every communication phase, this implies that, during communication phase *i* ($1 \leq i \leq T$), the sub-BAPs of all correct processors are synchronized within a real-time bound *B'*, with $B' \leq L_{i+1} / (1+\rho) - \tau_{max}$. Since *c* concludes communication phase *i* of its sub-BAP at or after time *t*, we may conclude that the earliest time at which *d* concludes communication phase *i* of its sub-BAP is at real-time *t* - *B*. We must prove that *m*(*mv*,(***p***;*d*)) always arrives in *d* in or before communication phase *i*+1 of *d*'s sub-BAP. Processor *d* concludes communication phase *i*+1 of its sub-BAP at or after $t + L_{i+1}(d) - B$. From A3.9, we know that, viewed from *d*, communication phase *i*+1 has a length of L_{i+1}. In real-time, this communication phase may be up to a factor (1+ρ) longer or shorter, depending on whether *d*'s processor clock runs slower or faster than real-time. Thus, in real-time, the minimum length of communication phase *i*+1 for *d* is $L_{i+1}(d) = L_{i+1} / (1+\rho)$. Hence, processor *d* concludes communication phase *i*+1 of its sub-BAP at or after $t + L_{i+1}(d) - B$, which is equal to $t + \tau_{max}$.

Thus, *m*(*mv*,(***p***;*d*)) always arrives in *d* in or before communication phase *i*+1 of *d*'s sub-BAP. This proves Lemma 3.6.3. ❑

THEOREM 3.6.
If a correct processor accepts a valid message with a certain message value *mv* during a BAP, then all correct processors will accept a valid message with message value *mv* before the end of this BAP.

Proof:
Assume that during execution of a BAP, some correct processor *c* accepts a valid message *m*(*mv*, (***p***;*c*)). Then, we must prove that any other correct processor *d* in the system also accepts a valid message with message value *mv*, before the end of the BAP. We will distinguish between the case that the length of path (***p***;*c*) is smaller than *T*+2, and the case that the length of path (***p***;*c*) is equal to *T*+2.

Case 1: the length of path (*p*;*c*) is smaller than *T*+2.
Assume that the length of path (***p***;*c*) is smaller than *T*+2. In this case, after having accepted message *m*(*mv*,(***p***;*c*)), processor *c* relays this message to all processors, the identification of which is not present in path (***p***;*c*). We must prove that any other correct processor *d* accepts a valid message with message value *mv* before the end of the BAP.

If the identification of processor *d* is not present in path (***p***;*c*), then processor *c* will relay message *m*(*mv*,(***p***;*c*)) to *d* immediately after having received message *m*(*mv*,(***p***;*c*)). From the description of the BAP, we know that a correct processor will only accept a valid message if the message is timely received. Since processor *c* has accepted *m*(*mv*,(***p***;*c*)), it must have timely received *m*(*mv*,(***p***;*c*)), i.e., in or before communication phase *i* ($1 \leq i \leq T$) of its sub-BAP, where the length of path (***p***;*c*) is greater than *i*. From Section 3.3.3.3., we know that relaying message *m*(*mv*,(***p***;*c*)) to *d* results in message *m*(*mv*,(***p***;*c*;*d*)) being sent to *d*. Processor *c* sends this message in communication phase *i* of its sub-BAP to *d*, hence, by Lemma 3.6.3., it will arrive in processor *d* in or before communication phase *i*+1 of *q*'s sub-BAP. Processor *d* will accept message *m*(*mv*,(***p***;*c*;*d*)), because the message is timely received in *d*, viz., the length of path (***p***;*c*;*d*) is greater than *i*+1. Hence, in this case, processor *d* accepts a valid message with message value *mv* before the end of the BAP.

If the identification of processor *d* is present in path (***p***;*c*), i.e. (***p***;*c*) = (***q***; *d*; ***r***; *c*) then, from the unforgeability properties UP1 and UP2, we know that processor *d* has accepted a valid message *m*(*mv*,(***q***;*d*)). Hence, in this case, processor *d* has also accepted a valid message with message value *mv* before the end of the BAP. ❑

Case 2: the length of path (*p;c*) is equal to *T*+2.
Assume that the length of path (***p***;*c*) is equal to *T*+2. Since *m*(*mv*,(***p***;*c*)) is a valid message, all processors, of which the identification is present in path (***p***;*c*), are different. Since it is assumed that the system contains at most *T* faulty processors, path (***p***;*c*) must contain the identification of at least 2 correct processors, hence, since processor *c* was assumed to be correct, path (***p***) must contain the identification of at least one correct processor. From the unforgeability properties UP1 and UP2, we may conclude that, either the source is correct, or at least one correct processor *x* must have received and accepted a valid message *m*(*mv*,(***q***;*x*)), with (***p***;*c*) = (***q***;*x*;***r***), and thus the length of path (***q***;*x*) is smaller than *T*+2.

If the source is correct, then it will have timely sent a valid message with message value *mv* to all other processors in the system, and it will have accepted a message with message value *mv* itself. From Theorem 3.5., we know that a valid message sent by a correct source to any correct processor, is timely received and accepted in that correct processor.

If some correct processor *x* has received and accepted a valid message *m*(*mv*,(***q***;*x*)) with message value *mv* and path information describing a path (***q***;*x*) with length smaller than *T*+2, then from case 1 above, it follows that all correct processors will accept a valid message with message value *mv* before the end of the BAP.

Hence, in both cases, any correct processor *d* will have received and accepted a valid message with message value *mv* before the end of the BAP. ❑

Hence, provided some correct processor *c* accepts during the BAP a valid message with message value *mv*, all other correct processors will also accept a valid message with message value *mv* during the BAP. This proves Theorem 3.6. ❑

With the help of Theorem 3.5. and 3.6., it is possible to prove that the authenticated self-synchronizing BAP presented in Section 3.3.6. satisfies the interactive consistency conditions IC1 and IC2, which have been stated in Section 3.1.2. as follows:

IC1. All correct processors agree on the initial value they think they have received from the source.
IC2. If the source is correct, then the above-mentioned agreement equals the initial value actually sent by the source.

The proof is given by the following theorem.

THEOREM 3.7.
Provided that the unforgeability properties UP1 and UP2 from Section 3.3.3. and assumptions A3.1. through A3.9. from Section 3.3.8. hold, the authenticated self-synchronizing BAP presented in Section 3.3.6. satisfies the interactive consistency conditions IC1 and IC2.

Proof:
We first prove that the authenticated self-synchronizing BAP satisfies IC1.

From Theorem 3.6., we know that if a correct processor accepts a valid message with a certain message value *mv* during the BAP, then all correct processors will accept a valid message with message value *mv* before the end of this BAP. Hence, the sets of message values of valid messages accepted during the BAP are equal for all correct processors.

The decision taken in a correct processor about the initial value sent by the source is based on applying a decision function on the set of message values of the valid messages accepted during the multicast process of the BAP. Since all correct processors apply the same decision function on the same set of message values, the decisions of all correct processors are the same, and thus, interactive consistency condition IC1 (the agreement condition) is satisfied.

We now prove that the authenticated self-synchronizing BAP satisfies IC2.

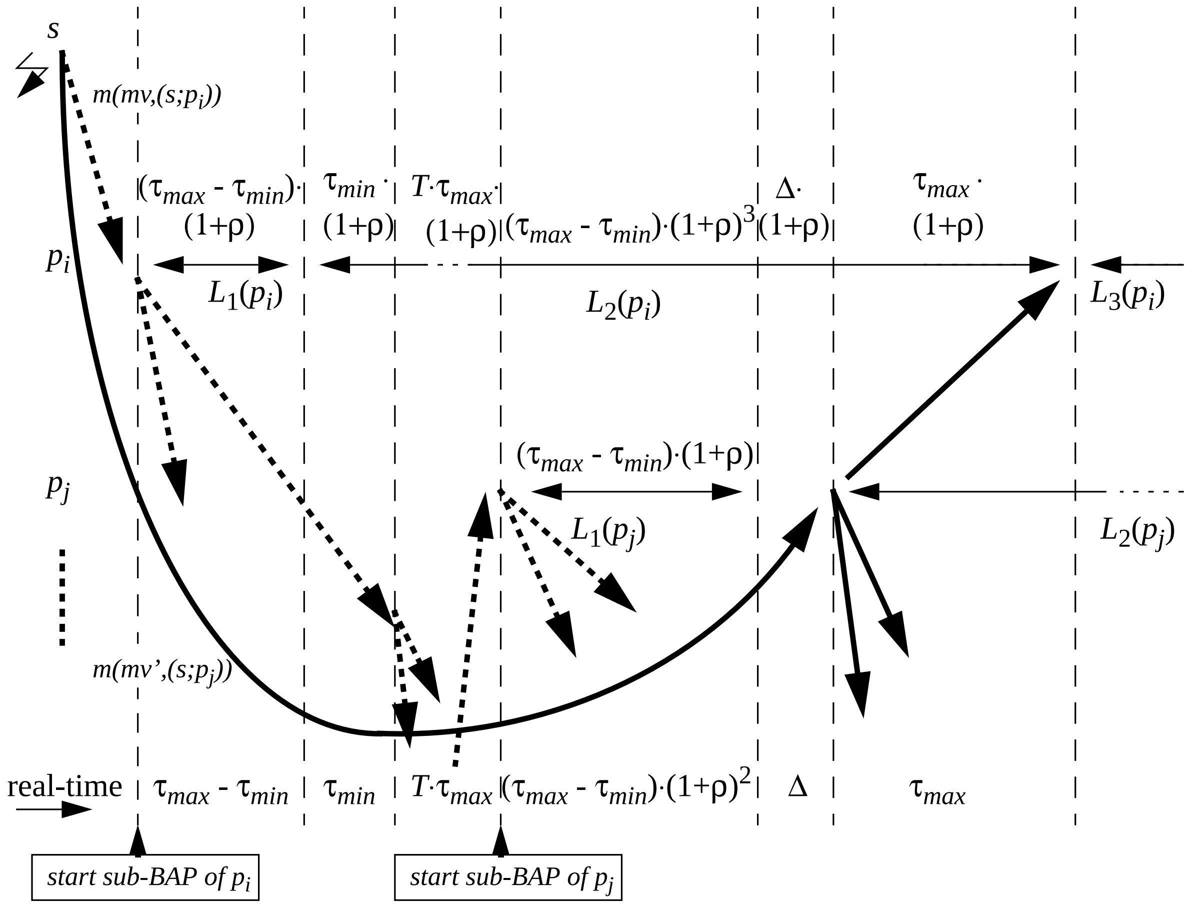
We assume that the source is correct. This implies that the source initiates the BAP by sending a valid message *m*(*mv*,(*s;x*)) to any other processor *x* in the system. Furthermore, it accepts a copy of the message itself in order to use it in the decision-making process of the BAP. From Theorem 3.5., we know that, if the source *s* which initiates the BAP is *correct*, any valid message sent directly from the source *s* to a correct processor *c* is timely received and accepted in processor *c*. Hence, any correct processor *c* will accept a valid message *m*(*mv*,(*s;c*)) received from the source *s*. For any correct processor *c*, the set of message values on which *c* will base its decision will thus at least contain message value *mv* of the message received from the source. However, the set of message values on which *c* bases its decision cannot contain other message values *mv*' ≠ *mv*, because the source *s* is correct and hence, only generates and multicasts valid messages with message value *mv*, and, by the unforgeability properties UP1 and UP2, faulty processors are not able to generate and multicast valid messages *m*(*mv*',(*s;c*)) with *mv*' ≠ *mv*, since such messages need the signature of processor *s*, which the faulty processors are unable to produce, since *s* is correct. Hence, any correct processor (including the source itself, since it has accepted a copy of its own message) will decide *mv*, and thus, interactive consistency condition IC2 (the validity condition) is satisfied.

This concludes the proof of Theorem 3.7. ❑

3.3.10. Proofs for optimized authenticated self-synchronizing BAPs

Assume a deterministic optimized authenticated self-synchronizing BAP is executed in a fully-connected synchronous system *N* consisting of *N* processors (including a source *s*), up to *T* of which may behave maliciously. Assume that assumptions A3.1 through A3.6 and A3.10 through A3.14 and the unforgeability properties UP1 and UP2 hold. We now prove that, in *N*, the interactive consistency conditions IC1 and IC2 are implied by assumptions A3.1 through A3.6 and A3.10 through A3.14. and the unforgeability properties UP1 and UP2.

In an example, in Figure 3.12. and 3.13., we indicate how protocol synchronization



$\rho \geq 0 \wedge \Delta \geq 0$

$(\forall\ p \in\ N : \forall\ k \in\ [1,T{+}1]{::}\ correct(p) \Rightarrow\ (1\ /\ (1{+}\rho))\cdot L_k(p) \leq L_k \leq (1{+}\rho)\cdot L_k(p))$

$L_1 = (\tau_{max} - \tau_{min})\cdot(1{+}\rho)$

$L_2 = (\tau_{max} - \tau_{min})\cdot(1{+}\rho)^3 + (\Delta + T\cdot\tau_{max} + \tau_{min})\cdot(1{+}\rho)$

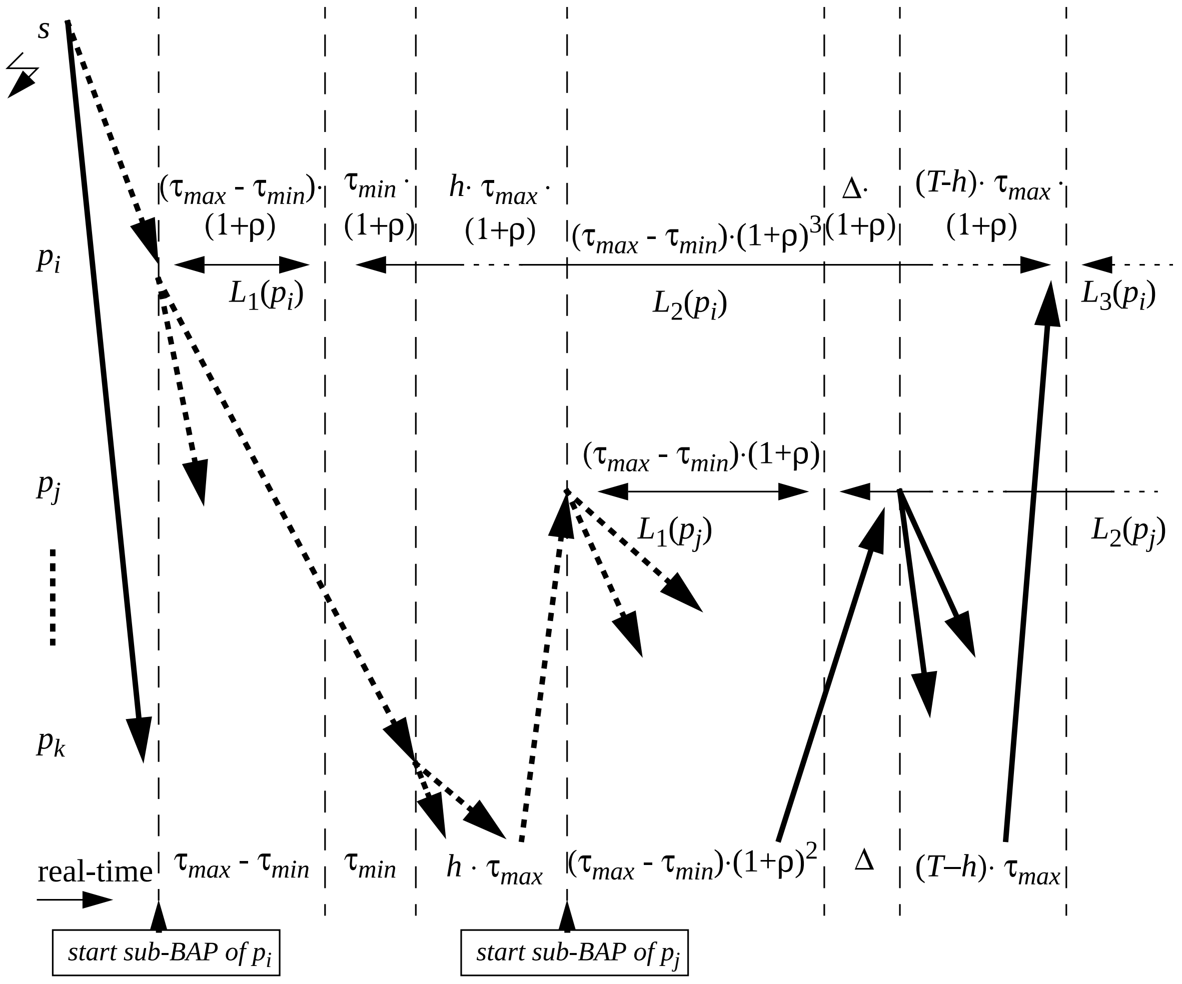
$(\forall\ k \in\ [3,T{+}1] :: L_k = [L_{k-1}\cdot(1{+}\rho) + \Delta]\cdot(1{+}\rho))$

For processor p_i and p_j it holds that

$(\forall\ k \in\ [1,T{+}1] :: (1/(1{+}\rho))\cdot L_k(p_i) = (1{+}\rho)\cdot\ L_k(p_j) = L_k)$

Processor p_j expects that the first valid message it receives has path information describing a path with length ≥ 2.

Figure 3.12. *Protocol synchronization between correct processors p_i and p_j in an optimized authenticated self-synchronizing BAP, in case the source s is faulty and $\rho \geq 0$ and $\Delta \geq 0$.*


s
$(\tau_{max} - \tau_{min})\cdot(1+\rho)$
$\tau_{min}\cdot(1+\rho)$
$h\cdot \tau_{max}\cdot(1+\rho)$
$(\tau_{max} - \tau_{min})\cdot(1+\rho)^3$
$\Delta\cdot(1+\rho)$
$(T\text{-}h)\cdot \tau_{max}\cdot(1+\rho)$
p_i
$L_1(p_i)$
$L_2(p_i)$
$L_3(p_i)$
$(\tau_{max} - \tau_{min})\cdot(1+\rho)$
p_j
$L_1(p_j)$
$L_2(p_j)$
p_k
real-time
$\tau_{max} - \tau_{min}$
τ_{min}
$h\cdot\tau_{max}$
$(\tau_{max} - \tau_{min})\cdot(1+\rho)^2$
Δ
$(T\text{–}h)\cdot \tau_{max}$
start sub-BAP of p_i
start sub-BAP of p_j

$\rho \geq 0 \wedge \Delta \geq 0$

$(\forall\ p \in\ N : \forall\ k \in\ [1,T+1] :: correct(p) \Rightarrow\ (1\ /\ (1+\rho))\cdot L_k(p) \leq L_k \leq (1+\rho)\cdot L_k(p))$

$L_1 = (\tau_{max} - \tau_{min})\cdot(1+\rho)$
$L_2 = (\tau_{max} - \tau_{min})\cdot(1+\rho)^3 + (\Delta + T\cdot\tau_{max} + \tau_{min})\cdot(1+\rho)$
$(\forall\ k \in\ [3,T+1] :: L_k = [L_{k-1}\cdot(1+\rho) + \Delta]\cdot(1+\rho))$

For processor p_i and p_j it holds that
$(\forall\ k \in\ [1,T+1] :: (1/(1+\rho))\cdot L_k(p_i) = (1+\rho)\cdot\ L_k(p_j) = L_k)$

Processor p_j expects that the first valid message it receives has path information describing a path with length $\geq h+2$ (with $1 \leq h \leq T$).

Figure 3.13. *Protocol synchronization between correct processors p_i and p_j in an optimized authenticated self-synchronizing BAP, in case the source s is faulty and $\rho \geq 0$ and $\Delta \geq 0$.*

between two correct processors p_i and p_j (p_i, $p_j \in N$) is achieved in an optimized authenticated self-synchronizing BAP, in the presence of a faulty source s ($s \in N$). Proofs for the general case are given by the lemma's and theorems in this section. In Figure 3.12. and 3.13., only valid messages that are expected, are depicted. (Invalid messages or messages that are not expected, will be rejected, and hence, do not influence the protocol. Therefore, these messages need not be depicted.) The time needed to communicate messages, the speeds at which the processor clocks of the different processors operate, and the clock value measurement uncertainties have been chosen such that the difference in protocol synchronization between processors p_i and p_j is maximal. The difference in protocol synchronization is maximal, if processor p_j expects that the first valid message it receives has path information describing a path with length greater than or equal to 2. See Figure 3.12. In this case, the path information of the message received by p_i at the end of $L_2(p_i)$ describes a path containing only 3 processors, and thus, this message just arrives in time in order to be accepted. As will be clear from Figure 3.13., if processor p_j expects that the first valid message it receives has path information describing a path with length greater than or equal to h+2 (with $1 \le h \le T$), then the maximal difference in protocol synchronization will be smaller than in the situation in Figure 3.12. In the situation in Figure 3.13, the path information of the message received by processor p_i at the end of $L_2(p_i)$ describes a path containing more than 3 processors, and thus the message arrives largely in time to be accepted.

LEMMA 3.8.1.
If a correct processor c receives the first valid message m of a BAP with path information describing a path containing n processors ($2 \le n \le T+1$), c knows that the BAP has been going on for at least τ_{min}.

Proof:
Follows directly from assumption A3.1. ❑

Again, notice that, although it is very well possible that the BAP is going on for longer than τ_{min}, c may not be able to conclude that (see also Section 3.3.9.).

THEOREM 3.8.
If the source which initiates the BAP is correct, then any valid message sent directly from the source to a correct processor c is timely received and accepted in processor c.

Proof:
Identical to the proof of Theorem 3.5., except for the fact that, at the start of the BAP, the source s does not send a valid message to all other correct processors in the system (as in Theorem 3.5.), but only to the correct processors in $S_{(s)}$, as selected by the subset selection rules of the BAP. ❑

LEMMA 3.9.1.
Once a correct processor p has concluded the first communication phase of its sub-BAP, then, within a real-time interval of $(\tau_{max} - \tau_{min}) \cdot (1+\rho)^2 + T \cdot \tau_{max} - (\tau_{max} - \tau_{min}) + \Delta$, all other correct processors have also concluded the first communication phase of their sub-BAP.

Proof:
Identical to the proof of Lemma 3.6.1., except for the fact that the maximum real-time difference between the start of the first communication phase of the sub-BAP of two different correct processors is not equal to τ_{max} (as in Lemma 3.6.1.) but equal to $T \cdot \tau_{max}$.

In an optimized authenticated self-synchronizing BAP, the above ***maximum*** real-time difference between the start of the first communication phase of the sub-BAP of two different correct processors occurs, if one of the correct processors, say *p*, is the source and initiates the BAP, and the subset selection rules are chosen such that the other correct processor, say *q*, only receives the first valid message from the BAP after *T* relay steps, and, in each relay step, a real-time interval of maximal length τ_{max} is needed to relay the message from the sender to the receiver of the message. While processor *p* starts the first communication phase of its sub-BAP as soon as it has initiated the BAP, processor *q* can only start the first communication phase of its sub-BAP a real-time period of $T \cdot \tau_{max}$ after *p* has initiated the BAP (i.e., as soon as processor *q* has received the first valid message of the BAP).

As in Lemma 3.6.1., the maximal real-time difference between the end of the first communication phase of the sub-BAPs of two different correct processors *p* and *q* is determined by the maximum real-time difference between the start of the first communication phase of the sub-BAP of processor *p* and *q*, and by the maximum real-time difference in the length of the first communication phase in the sub-BAPs of processor *p* respectively, of processor *q*.

By A3.14., for any correct processor *c*, the length L_1 of the first communication phase of *c*'s sub-BAP, as measured by *c*'s processor clock, is equal to $(\tau_{max} - \tau_{min}) \cdot (1+\rho)$. This value for L_1 is identical to the value derived for L_1 in Lemma 3.6.1. Hence, the maximal and minimal real-time length of the first communication phase of the sub-BAP of a correct processor are as given in Lemma 3.6.1.

In a similar way as it is done in Lemma 3.6.1., we will now calculate the maximal real-time difference between the end of the first communication phase of the sub-BAPs of two different correct processors.

The real-time difference between the end of the first communication phase of the sub-BAPs of two different correct processors is ***maximal***, if the real-time difference between the start of the first communication phase is maximal, and the real-time difference in lengths of the first communication phase of the sub-BAPs in the different processors is maximal, i.e., if the length of the first communication phase of the sub-BAP of the correct processor that starts its sub-BAP first is minimal, whereas the length of the first communication phase of the sub-BAP of the correct processor that starts its sub-BAP $T \cdot \tau_{max}$ later is maximal.

Let *p* be the first correct processor that starts its sub-BAP, say at time $t_{earliest\_start}$. Then, the earliest time at which processor *p* can conclude the first communication phase of its sub-BAP is at time $t_{earliest\_end} = t_{earliest\_start} + (\tau_{max} - \tau_{min})$ (i.e., if the real-time length of the first communication phase of *p*'s sub-BAP is minimal). Then, the latest moment at which any other correct processor q can start the first communication phase of its sub-BAP, is at real-time $t_{latest\_start} = t_{earliest\_start} + T \cdot \tau_{max}$ (i.e., in case the subset selection rules are chosen such that q only receives the first valid message of the BAP after T relay steps, and, in each relay step, a real-time interval of maximal length τ_{max} is needed to relay the message from the sender to the receiver). The latest moment at which q can conclude the first communication phase of its sub-BAP is at $t_{latest\_end} = t_{latest\_start} + (\tau_{max} - \tau_{min}) \cdot (1+\rho)^2 + \Delta$ (i.e., in case the real-time length of the first communication phase of q's sub-BAP is maximal).

Hence, as soon as a correct processor has concluded the first communication phase of its sub-BAP, after a real-time interval of at most

$$t_{latest\_end} - t_{earliest\_end} =$$
$$(t_{earliest\_start} + T \cdot \tau_{max} + (\tau_{max} - \tau_{min}) \cdot (1+\rho)^2 + \Delta) - (t_{earliest\_start} + (\tau_{max} - \tau_{min})) =$$
$$(\tau_{max} - \tau_{min}) \cdot (1+\rho)^2 + T \cdot \tau_{max} - (\tau_{max} - \tau_{min}) + \Delta,$$

all other correct processors have also concluded the first communication phase of their sub-BAP. This concludes Lemma 3.9.1. ❑

LEMMA 3.9.2.
After communication phase i ($1 \le i \le T$), in real-time, the sub-BAPs of all correct processors are synchronized within a real-time bound of $(L_{i+1} / (1+\rho)) - \tau_{max}$.

Proof:
The proof is similar to that of Lemma 3.6.2. However, notice that, in Lemma 3.9.2., the lengths of the communication phases of the BAP as measured by the processor clock of any correct processor are given by A3.14 instead of by A3.9. (as in Lemma 3.6.2.)

Lemma 3.9.2. is proved by induction on i.

Basis: i = 1.
From Lemma 3.9.1. we know that after communication phase 1, the sub-BAPs of all correct processors have been synchronized within a real-time bound of $(\tau_{max} - \tau_{min}) \cdot (1+\rho)^2 + T \cdot \tau_{max} - (\tau_{max} - \tau_{min}) + \Delta$, which is equal to $(L_2 / (1+\rho)) - \tau_{max}$.

Induction step: i > 1.
We assume that Lemma 3.9.2. holds for $j = i - 1$. Thus, the induction hypothesis is:
After communication phase $i - 1 (2 \le i \le T)$, in real-time, the sub-BAPs of all correct processors are synchronized within a real-time bound of $L_i / (1+\rho) - \tau_{max}$.

Now, we prove that Lemma 3.9.2. also holds for $j = i$.

Notice that the maximal real-time difference in synchronization occurs between the sub-BAP of a correct processor s which runs a factor $(1+\rho)$ slower than real-time and that of a correct processor f which runs a factor $(1+\rho)$ faster than real-time. The real-time difference is maximal if processor f immediately detects the end of its communication phase, whereas processor s only detects it after real-time Δ.

For all i (with $1 \le i \le T+1$) and any correct processor p, let $L_i(p)$ be the real-time length of communication phase *i* in *p*. For a correct processor *s* that runs a factor $(1+\rho)$ slower than real-time, the maximum real-time length of communication phase *i* in *s* is $L_i(s) = L_i \cdot (1+\rho) + \Delta$. For a correct processor *f* that runs a factor $(1+\rho)$ faster than real-time, the minimum real-time length of communication phase *i* in *f* is $L_i(f) = L_i / (1+\rho)$. So, the real-time difference in protocol synchronization increases by an amount less than or equal to $L_i(s) - L_i(f)$. From the induction hypothesis, we know that the real-time difference after communication phase $i - 1$ is less than or equal to $L_i / (1+\rho) - \tau_{max}$, which is equal to $L_i(f) - \tau_{max}$. After communication phase *i*, the maximal real-time difference in protocol synchronization is $(L_i(f)-\tau_{max})+(L_i(s)-L_i(f)) = L_i(s) - \tau_{max}$, which is equal to $L_i \cdot (1+\rho) + \Delta - \tau_{max}$. Since, by A3.14, for $i > 1$, $L_{i+1} = [L_i \cdot (1+\rho) + \Delta]\cdot (1+\rho)$, the maximal real-time difference in protocol synchronization after communication phase *i* is $L_i \cdot (1+\rho) + \Delta - \tau_{max} = L_{i+1} / (1+\rho) - \tau_{max}$. This proves Lemma 3.9.2. ❑

LEMMA 3.9.3.
Let $m(mv,(\underline{\mathbf{p}}))$ be any valid message which is timely received and accepted by a correct processor *c* in communication phase *i* ($1 \leq i \leq T$) of *c*’s sub-BAP and assume that *c* relays $m(mv,(\underline{\mathbf{p}}))$ to a correct processor *d*, i.e., processor *c* sends $m(mv,(\underline{\mathbf{p}};d))$ to *d*. Then, message $m(mv,(\underline{\mathbf{p}};d))$ arrives in processor *d* in or before communication phase $i+1$ of *d*’s sub-BAP.

Proof:
Identical to the proof of Lemma 3.6.3., except for the fact that the lengths of the communication phases of the BAP, as measured by the processor clocks of any correct processor are given by A3.14, instead of by A3.9 (as is the case in Lemma 3.6.3.) ❑

THEOREM 3.9.
If a correct processor accepts a valid message with a certain message value *mv* during a BAP, then all correct processors will accept a valid message with message value *mv* before the end of this BAP.

Proof:
Assume that during execution of a BAP, some correct processor *c* accepts a valid message $m(mv, (\underline{\mathbf{p}};c))$. Then, we must prove that any other correct processor *d* in the system also accepts a valid message with message value *mv*, before the end of the BAP. We will distinguish between the case that the length of path $(\underline{\mathbf{p}};c)$ is smaller than $T+2$, and the case that the length of path $(\underline{\mathbf{p}};c)$ is equal to $T+2$.

Case 1: the length of path $(\underline{\mathbf{p}};c)$ is smaller than $T+2$.
Assume that the length of path $(\underline{\mathbf{p}};c)$ is smaller than $T+2$. In this case, after having accepted message $m(mv,(\underline{\mathbf{p}};c))$, processor *c* relays this message to all processors in set $S_{(\underline{\mathbf{p}};c)}$, i.e. the subset of processors selected by the subset selection rules of the BAP. We must prove that any other correct processor *d* accepts a valid message with message value *mv* before the end of the BAP.

If processor *d* is an element of set $S_{(\underline{\mathbf{p}};c)}$ (i.e. *c* should relay $m(mv,(\underline{\mathbf{p}};c))$ to *d* according to the subset selection rules of the BAP), then processor *c* will relay message $m(mv,(\underline{\mathbf{p}};c))$ to *d* immediately after having received message $m(mv,(\underline{\mathbf{p}};c))$. From the description of the BAP, we know that a correct processor will only accept a valid message if the message is timely received. Since processor c has accepted $m(mv,(\underline{\boldsymbol{p}};c))$, it must have timely received $m(mv,(\underline{\boldsymbol{p}};c))$, i.e., in or before communication phase i ($1 \leq i \leq T$) of its sub-BAP, where the length of path $(\underline{\boldsymbol{p}};c)$ is greater than i. From Section 3.3.3.3., we know that relaying message $m(mv,(\underline{\boldsymbol{p}};c))$ to d results in message $m(mv,(\underline{\boldsymbol{p}};c;d))$ being sent to d. Processor c sends this message in communication phase i of its sub-BAP to d, hence, by Lemma 3.6.3., it will arrive in processor d in or before communication phase i+1 of q's sub-BAP. Processor d will accept message $m(mv,(\underline{\boldsymbol{p}};c;d))$, because the message is timely received in d, viz., the length of path $(\underline{\boldsymbol{p}};c;d)$ is greater than i+1. Hence, in this case, processor d accepts a valid message with message value *mv* before the end of the BAP.

If processor d is not an element of set $S_{(\underline{\boldsymbol{p}};c)}$, then we distinguish between two cases: first, we consider the case that the identification of processor d is present in path $(\underline{\boldsymbol{p}};c)$, then we consider the case that the identification of processor d is not present in path $(\underline{\boldsymbol{p}};c)$.

First, consider the case that processor d is not an element of set $S_{(\underline{\boldsymbol{p}};c)}$, but the identification of processor d is present in path $(\underline{\boldsymbol{p}};c)$, i.e. $(\underline{\boldsymbol{p}};c) = (\underline{\boldsymbol{q}};\ d;\ \underline{\boldsymbol{r}};\ c)$. Then, from the unforgeability properties UP1 and UP2, we know that processor d has accepted a valid message $m(mv,(\underline{\boldsymbol{q}};d))$. Hence, in this case, processor d has accepted a valid message with message value *mv* before the end of the BAP.

Now consider the case that processor d is not an element of set $S_{(\underline{\boldsymbol{p}};c)}$, and the identification of processor d is not present in path $(\underline{\boldsymbol{p}};c)$. The fact that processor c has accepted message $m(mv,(\underline{\boldsymbol{p}};c))$ implies that c did expect $m(mv,(\underline{\boldsymbol{p}};c))$ according to the subset selection rules of the BAP. According to A3.12., the subset selection rules are chosen such that every valid message from the *source* is relayed to all correct processors within T relay steps. Since the identification of d is not present in path $(\underline{\boldsymbol{p}};c)$, the subset selection rules guarantee that message $m(mv,(\underline{\boldsymbol{p}};c))$ relayed by c will (indirectly) be relayed to d within $(T + 2 - x)$ relay steps, in which x is the length of path $(\underline{\boldsymbol{p}};c)$. Hence, d will timely receive and accept a message with message value *mv* before the end of the BAP. ❑

Case 2: the length of path $(\underline{\boldsymbol{p}};\boldsymbol{c})$ is equal to $\boldsymbol{T}$+2.

Assume that the length of path $(\underline{\boldsymbol{p}};c)$ is equal to T+2. Since $m(mv,(\underline{\boldsymbol{p}};c))$ is a valid message, all processors, of which the identification is present in path $(\underline{\boldsymbol{p}};c)$, are different. Since it is assumed that the system contains at most T faulty processors, path $(\underline{\boldsymbol{p}};c)$ must contain the identification of at least 2 correct processors, hence, since processor c was assumed to be correct, path $(\underline{\boldsymbol{p}})$ must contain the identification of at least one correct processor. From the unforgeability properties UP1 and UP2, we may conclude that, either the source is correct, or at least one correct processor x must have received and accepted a valid message $m(mv,(\underline{\boldsymbol{q}};x))$, with $(\underline{\boldsymbol{p}};c) = (\underline{\boldsymbol{q}};x;\underline{\boldsymbol{r}})$, and thus the length of path $(\underline{\boldsymbol{q}};x)$ is smaller than T+2.

If the source is correct, then, by A3.12, every correct processor in the system will have timely received a valid message from the source with message value *mv* within at most T relay steps, and the source will have accepted a message with message value *mv* itself. From Theorem 3.5., we know that a valid message sent by a correct source to any correct processor, is timely received and accepted in that correct processor.

If some correct processor *x* has received and accepted a valid message *m*(*mv*,(***q***;*x*)) with message value *mv* and path information describing a path (***q***;*x*) with length smaller than *T*+2, then from case 1 above, it follows that all correct processors will accept a valid message with message value *mv* before the end of the BAP.

Hence, in both cases, any correct processor *d* will have received and accepted a valid message with message value *mv* before the end of the BAP. ❏

Hence, provided some correct processor *c* accepts during the BAP a valid message with message value *mv*, all other correct processors will also accept a valid message with message value *mv* during the BAP. This proves Theorem 3.9. ❏

With the help of Theorem 3.8. and 3.9., it is possible to prove that the optimized authenticated self-synchronizing BAP presented in Section 3.3.7. satisfies the interactive consistency conditions IC1 and IC2, which have been stated in Section 3.1.2. as follows:

IC1. All correct processors agree on the initial value they think they have received from the source.

IC2. If the source is correct, then the above-mentioned agreement equals the initial value actually sent by the source.

The proof is given by the following theorem.

THEOREM 3.10.
Provided that the unforgeability properties UP1 and UP2 from Section 3.3.3. and assumptions A3.1. through A3.6. and A3.10 through A3.14. from Section 3.3.8. hold, the optimized authenticated self-synchronizing BAP presented in Section 3.3.7. satisfies the interactive consistency conditions IC1 and IC2.

Proof:
We first prove that the authenticated self-synchronizing BAP satisfies IC1.

From Theorem 3.9., we know that if a correct processor accepts a valid message with a certain message value *mv* during the BAP, then all correct processors will accept a valid message with message value *mv* before the end of this BAP. Hence, the sets of message values of valid messages accepted during the BAP are equal for all correct processors.

The decision taken in a correct processor about the initial value sent by the source is based on applying a decision function on the set of message values of the valid messages accepted during the multicast process of the BAP. Since all correct processors apply the same decision function on the same set of message values, the decisions of all correct processors are the same, and thus, interactive consistency condition IC1 (the agreement condition) is satisfied.

We now prove that the authenticated self-synchronizing BAP satisfies IC2.

We assume that the source is correct. This implies that the source initiates the BAP by sending a valid message $m(mv,(s;x))$ to any processor x in the set $S_{(s)}$, i.e. the set of processors to which the source should send a message, according to the subset selection rules of the BAP. From A3.12., we know that the subset selection rules are chosen such that any message from the source will be relayed to any correct processor in the system within at most T relay steps.

Furthermore, the source accepts a copy of the message itself in order to use it in the decision-making process of the BAP. From Theorem 3.8., we know that, if the source s which initiates the BAP is *correct*, any valid message sent directly from the source s to any correct processor c in $S_{(s)}$ is timely received and accepted in processor c. Furthermore, from A3.12., we know that the subset selection rules are selected such that every valid message from the source is relayed to all correct processors in the system within at most T relay steps. Hence, before the end of the BAP, any correct processor c will accept a valid message $m(mv,(s;\underline{\mathbf{p}};c))$ received (directly or indirectly) from the source s.

For any correct processor c, the set of message values on which c will base its decision will thus at least contain message value mv of the message received from the source. However, the set of message values on which c bases its decision cannot contain other message values $mv' \neq mv$, because the source s is correct and hence, only generates and multicasts valid messages with message value mv, and, by the unforgeability properties UP1 and UP2, faulty processors are not able to generate and multicast valid messages $m(mv',(s;\underline{\mathbf{p}};c))$ with $mv' \neq mv$, since such messages need the signature of processor s, which the faulty processors are unable to produce, since s is correct. Hence, any correct processor (including the source itself, since it has accepted a copy of its own message) will decide mv, and thus, interactive consistency condition IC2 (the validity condition) is satisfied.

This concludes the proof of Theorem 3.10. ❑

3.3.11. A comparison of authenticated self-synchronizing BAPs and optimized authenticated self-synchronizing BAPs

In this section, for some practical values of T, N, ρ, τ_{min}, and τ_{max}, we compare the optimized authenticated self-synchronizing BAPs (described in Section 3.3.7.) with the authenticated self-synchronizing BAPs (described in Section 3.3.6.), which we will refer to as non-optimized self-synchronizing BAPs[45].

We choose $\rho=10^{-4}$, $\tau_{min}= 0$ ms and $\tau_{max} = 20$ ms. These values are realistic for an ATM-network. Furthermore, we choose $\Delta= 20$ ms.

In Table 3.7 and 3.8, for some values of T and N, the required number of messages for both non-optimized and optimized authenticated self-synchronizing BAPs are given. We assume that, in our optimized authenticated self-synchronizing BAPs, in every

45. We do not compare the optimized self-synchronizing BAPs to other BAPs discussed in this chapter, since the latter BAPs are based on a system model with more restrictive synchronicity assumptions.

communication phase except the last one, a valid message is relayed to a subset of T+1 processors.

For a non-optimized authenticated self-synchronizing BAP, the required number of messages *mess* is given by [46]

$$mess = \sum_{i=1}^{T+1} i! \cdot \begin{bmatrix} N-1 \\ i \end{bmatrix}$$

whereas for an optimized authenticated self-synchronizing BAP, for $N \geq 2T + 1$, the required number of messages *mess* is given by [47]

$$mess = \left(\left(\sum_{i=1}^{T} (T+1)^{i} \right) + (T+1)^{T} \cdot (N-T-1) \right)$$

and for $T+2 \leq N \leq 2T$, by

$$\left(\sum_{i=1}^{N-T-1} (T+1)^{i} \right) +$$
$$\left(\sum_{i=N-T}^{T+1} (T+1)^{N-T-1} \cdot (i-N+T+1)! \cdot \begin{bmatrix} T \\ i-N+T+1 \end{bmatrix} \right)$$

The results show that for $N > T$+2, optimized authenticated self-synchronizing BAPs require fewer messages than non-optimized ones.

In Table 3.9, for some values of T, we have compared the maximum protocol execution time of both types of BAPs. We see that for $T \geq 2$, in an optimized BAP, the maximum execution time of a sub-BAP is greater than in a non-optimized BAP. The maximum algorithm execution time of either BAP is given by [48]

$$maxexecutiontime = (1+\rho) \cdot \sum_{k=1}^{T+1} L_k$$

where L_k is the length of communication phase k ($1 \leq k \leq T$+1) of the sub-BAP (of the corresponding type of BAP) of a certain processor p, as viewed from p.

We see that for $T \geq 2$, in an optimized BAP, the execution time of a sub-BAP is greater

46. The BAP is a self-synchronizing version of the Lamport-protocol. See also Section 3.2.4.2.
47. The BAP is a self-synchronizing version of the Minimum Direction protocol presented in Section 3.2.3.1. and 3.2.4.4.
48. The protocol execution time *executiontime* measured by a processor clock of any correct processor is equal to

$$executiontime = \sum_{k=1}^{T+1} L_k$$

However, since the processor clock of a correct processor may run up to a factor (1+ρ) slower than real-time, the maximum protocol execution time *maxexecutiontime* is equal to *executiontime*·(1+ρ).

than in a non-optimized BAP.

Table 3.7: Required number of messages for T=1

N	#mess. in non-optimized BAPs	#mess. in optimized BAPs
3	4	4
4	9	6
5	16	8
16	256	30

Table 3.8: Required number of messages for T=2

N	#mess. in non-optimized BAPs	#mess. in optimized BAPs
4	15	15
5	40	30
6	85	39
16	2955	129

Table 3.9: Protocol execution time of sub-BAP (in ms)

T	Execution time in non-optimized BAPs	Execution time in optimized BAPs
1	80.020	80.020
2	160.052	200.064
3	260.104	380.152
4	380.180	620.300

3.3.12. Conclusion

The authenticated self-synchronizing Byzantine Agreement Protocols described in this section overcome certain restrictions regarding the synchronicity of the system that are required by existing authenticated BAPs in literature. We have also described optimized authenticated self-synchronizing BAPs, that require lower communication overhead than the authenticated self-synchronizing BAPs (for $N > T + 2$), be it at the cost of increased lengths of the communication phases of the protocol (for $T > 1$).

We have proved that, provided that the assumptions A3.1 through A3.9 (in Section 3.3.8.) and the unforgeability properties UP1 and UP2 (from Section 3.3.3.) hold, execution of an authenticated self-synchronizing BAP in a fully-connected system of N processors, up to T of which may behave maliciously, guarantees Byzantine Agreement, i.e., satisfaction of the interactive consistency conditions IC1 and IC2. Provided that the assumptions A3.1 through A3.6 and A3.10 through A3.14 (in Section 3.3.8.) and the unforgeability properties UP1 and UP2 (from Section 3.3.3.) hold, the same argument holds for our optimized authenticated self-synchronizing BAPs.

3.4. Authenticated self-synchronizing interactive consistency algorithms

As stated in Section 3.1.7., in distributed systems, there exists a need for ICAs that work without requiring the correct processors to reach exact agreement about a common point in time. The ***authenticated self-synchronizing ICAs*** and ***optimized authenticated self-synchronizing ICAs*** which will be introduced in Section 3.4.1. and 3.4.2. respectively, can be used for this purpose, since they do not require the correct processors to reach exact agreement about a common point in time. These ICAs consist of execution of N authenticated self-synchronizing BAPs (introduced in Section 3.3.6.) respectively N optimized authenticated self-synchronizing BAPs (introduced in Section 3.3.7.), and work in a synchronous system, with arbitrary nonnegative ρ and τ_{max}, and arbitrary nonnegative $\tau_{min} \leq \tau_{max}$, in which the processors are not synchronized and the start of the ICA is not a priori known. In both types of self-synchronizing ICAs, the processors should inform each other about the start of the ICA, and protocol synchronization between the different BAPs that together make up the ICA should explicitly be established.

A self-synchronizing ICA differs only slightly from a self-synchronizing BAP. The ICAs presented in this section have been described in a similar way as the BAPs in Section 3.3.6. and 3.3.7. respectively, in order to emphasize the similarities and differences between both types of algorithms.

In Section 3.4.1., an event-based description of an authenticated self-synchronizing ICA is given, followed (in Section 3.4.2.) by a description of an optimized authenticated self-synchronizing ICA. In Section 3.4.3., both types of ICAs are compared with regard to the required number of messages and the algorithm execution time.

3.4.1. A description of an authenticated self-synchronizing ICA

An authenticated self-synchronizing ICA has similarities with the authenticated self-synchronizing BAP given in Section 3.3.6. In fact, an ICA runs on a system, N, of N processors, and consists of execution of N BAPs. Every processor $p \in N$ is the source in one of the BAPs belonging to the ICA. Of these BAPs, for any processor $p \in N$, we will denote the BAP in which processor p acts as the source as BAP_p.

Before we give the description of the authenticated self-synchronizing ICA, we make the following remarks.

- In a self-synchronizing ICA, the processors do not know a priori when the ICA starts. It is assumed that at least one processor, say p, is triggered to initiate the ICA (e.g., due to the occurrence of some external event, or to its processor clock reaching a certain predefined value). This processor p initiates the ICA by starting its own BAP_p which is part of the ICA (i.e., processor p acts as the source and generates and sends a message $m(mv,(p;k))$ to every processor $k \in N \setminus \{i\}$). Simultaneously, processor p starts the first communication phase of its sub-BAP of any other BAP_j ($j \in N \setminus \{p\}$).

- The other processors are unaware of the start of the ICA until they receive the first valid message of that ICA. At receipt of this message, say from BAP$_p$, any processor $i \in N \setminus \{p\}$ will participate in BAP$_p$ (by starting the first communication phase of its sub-BAP of BAP$_p$). Simultaneously, processor i will also start the first communication phase of its sub-BAP of any other BAP$_j$ ($j \in N \setminus \{p\}$). Processor i starts its own BAP$_i$ by generating and sending a message $m(mv,(i;k))$ to every processor $k \in N \setminus \{i\}$.

- An ICA consists of execution of a number of BAPs on a set of processors. In its turn, each BAP consists of a set of sub-BAPs each running on a different processor. The set of all sub-BAPs of the BAPs belonging to an ICA and running on a certain processor p ($p \in N$) will be referred to as the ***sub-ICA of processor p***.

- Since every processor $p \in N$ starts all its sub-BAPs of the BAPs belonging to an ICA simultaneously, it is justified to speak of communication phase j ($1 \le j \le T+1$) of the sub-ICA of processor p.

- In order to be able to decide whether or not a valid message m of a certain BAP$_s$ belonging to the ICA was received in time, according to D3.25, a correct processor that receives m in communication phase j ($1 \le j \le T+1$) of its sub-ICA, checks if j is smaller than the number of processors in the path information of m. We assume, that every processor $i \in N$ stores the clock values at which the communication phases of its sub-ICA end in an array $endcommphase_i$.

  Initially, the processors do not know when an ICA will start, and hence, any processor $i \in N$ does not know these clock values until i receives the first valid message from the ICA. Hence, for any processor $i \in N$, and any communication phase j of i's sub-ICA, with $1 \le j \le T+1$, $endcommphase_i(j)$ is initialized to ∞.

  As soon as processor i receives the first valid message from the ICA, say at clock value $C(i,t)$ of its processor clock, i starts the first communication phase of its sub-ICA. Then, processor i is able to calculate at which clock values $C(i,t')$ of its processor clock each of the $T+1$ communication phases of its sub-ICA ends, since, as shown below, i knows a priori the length L_k ($1 \le k \le T+1$) of every communication phase of its sub-ICA. For all h, with $1 \le h \le T+1$, it holds that

$$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

  By A3.15., for all k, with $1 \le k \le T+1$, L_k is the length of communication phase k of i's sub-ICA, as viewed from i, where L_k is recursively defined by:

$$L_1 = 2\tau_{max} \cdot (1+\rho)$$

$$L_2 = 2\tau_{max} \cdot (1+\rho)^3 + \Delta \cdot (1+\rho)$$

$$(\forall\, h \in [3,T+1] :: L_h = [L_{h-1} \cdot (1+\rho) + \Delta] \cdot (1+\rho))$$

- If all BAPs that are part of the ICA should have terminated according to a certain processor $i \in N$, i decides on the outcome of the ICA on basis of the decisions calculated in the different BAPs of the ICA.

- It is assumed that processor *i* can distinguish between messages belonging to different ICAs. In the ICA described below, we will omit the identification of the ICA from the messages, since this unnecessarily complicates the description of the algorithm.

An authenticated self-synchronizing ICA can be described as follows:
Initially, for each processor $i \in N$, for all j, with $1 \le j \le T+1$, $endcommphase_i(j) = \infty$

For some (possibly more than one) processor $s \in N$:
Event: *Processor s is triggered to start the ICA at time C(s,t).*
Action: *Processor s acts as the source in BAP_s belonging to the ICA and generates and sends a message m(mv,(s;k)) to every processor $k \in N \setminus \{s\}$. The source accepts a copy of its own message in order to use it in the decision-making process of BAP_s.*
Simultaneously, the source starts the first communication phase of its sub-ICA. Furthermore, the source calculates at which clock values C(s,t') of its processor clock each of the T+1 communication phases of its sub-ICA ends. For all h, with $1 \le h \le T+1$, it holds that

$$endcommphase_s(h) = C(s,t) + \sum_{k=1}^{h} L_k$$

For each processor $i \in N$:
Event: *$C(i,t) \ge endcommphase_i(T+1)$*
Action: *processor i decides the outcome of the ICA on basis of the decisions from all of the BAP_k (for any $k \in N$) it has calculated.*

Event: *message $mess_j(i)$ (with $1 \le j \le M_i$) received at time C(i,t)*
Action:
1. *Check if $mess_j(i)$ is a valid message, i.e., check if $mess_j(i) \in$ ValidMessages.*
   If invalid($mess_j(i)$) then reject $mess_j(i)$ and abort, i.e., do no perform further actions on $mess_j(i)$.
   If valid($mess_j(i)$), then $mess_j(i)$ = m(mv,($\mathbf{\underline{q}}$)) of some BAP_s belonging to the ICA (with s = first(($\mathbf{\underline{q}}$)), for some path ($\mathbf{\underline{q}}$) $\in$ ValidPaths and some mv $\in$ MessageValues. If last(($\mathbf{\underline{q}}$)) $\ne$ i, then reject $mess_j(i)$ and abort, otherwise $mess_j(i)$ = m(mv,($\mathbf{\underline{p}}$;i)) for path ($\mathbf{\underline{p}}$) $\in$ ValidPaths with ($\mathbf{\underline{p}}$) = prefix(($\mathbf{\underline{q}}$)).
2. *Check if a valid message $mess_k(i)$ (with $1 \le k < j$) with path($mess_k(i)$) = ($\mathbf{\underline{p}}$;i) has already been accepted.*
   If such a message has already been accepted, then reject $mess_j(i)$ and abort.
3. *Check if m(mv,($\mathbf{\underline{p}}$;i)) has been received in time, i.e., check if C(i,t) $\le$ $endcommphase_i$(length(($\mathbf{\underline{p}}$;i))-1). If m(mv,($\mathbf{\underline{p}}$;i)) has been received in time, i accepts m(mv,($\mathbf{\underline{p}}$;i)), otherwise i rejects m(mv,($\mathbf{\underline{p}}$;i)).*
4. *If m(mv,($\mathbf{\underline{p}}$;i)) is the first valid message of the ICA that i receives (and hence, accepts), i starts the first communication phase of its sub-ICA at receipt of m(mv,($\mathbf{\underline{p}}$;i)). Processor i now calculates at which clock*

values C(i,t') of its processor clock each of the T+1 communication phases of its sub-ICA ends. For all h, with 1 ≤ h ≤ T+1, it holds that

$$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

Furthermore, processor i is triggered to start its own BAP (BAP_i) belonging to the ICA. Hence, processor i acts as the source in BAP_i belonging to the ICA and generates and sends a message m(mv,(i;k)) to every processor k ∈ N \ {i}. Processor i accepts a copy of its own message in order to use it in the decision-making process of BAP_i.

5. *If i accepted m(mv,(**p**;i)) and path(m(mv,(**p**;i))) contains fewer than T+2 processors, for every processor h that is not in path(m(mv,(**p**;i))), i signs message m(mv,(**p**;i)) and i relays m(mv,(**p**;i)) to h, immediately after m(mv,(**p**;i)) was received by i. As stated in Section 3.3.3.3., relaying message m(mv,(**p**;i)) to any processor h results in m(mv,(**p**;i;h)) being sent to h.*

All correct processors are assumed to execute the above-described algorithm. The behaviour of faulty processors, however, may deviate arbitrarily from the described algorithm.

Assume an authenticated self-synchronizing ICA is executed in a fully-connected synchronous system *N* consisting of *N* processors (including a source *s*), up to *T* of which may behave maliciously. We assume that for any authenticated self-synchronizing BAP belonging to the ICA, the unforgeability properties UP1 and UP2 hold, and the assumptions A3.1 through A3.3, A3.5 through A3.8, and A3.15 and A3.16 are satisfied, where A3.15 and A3.16 are given by

A3.15. Every processor in the system may have a different view of time and of the start and end of communication phases in the ICA. For all *i*, with $1 \le i \le T+1$, for any correct processor $p \in N$, let L_i be the length of communication phase *i* of *p*'s sub-ICA, as viewed from *p*. Then:

$$L_1 = 2\tau_{max} \cdot (1+\rho)$$

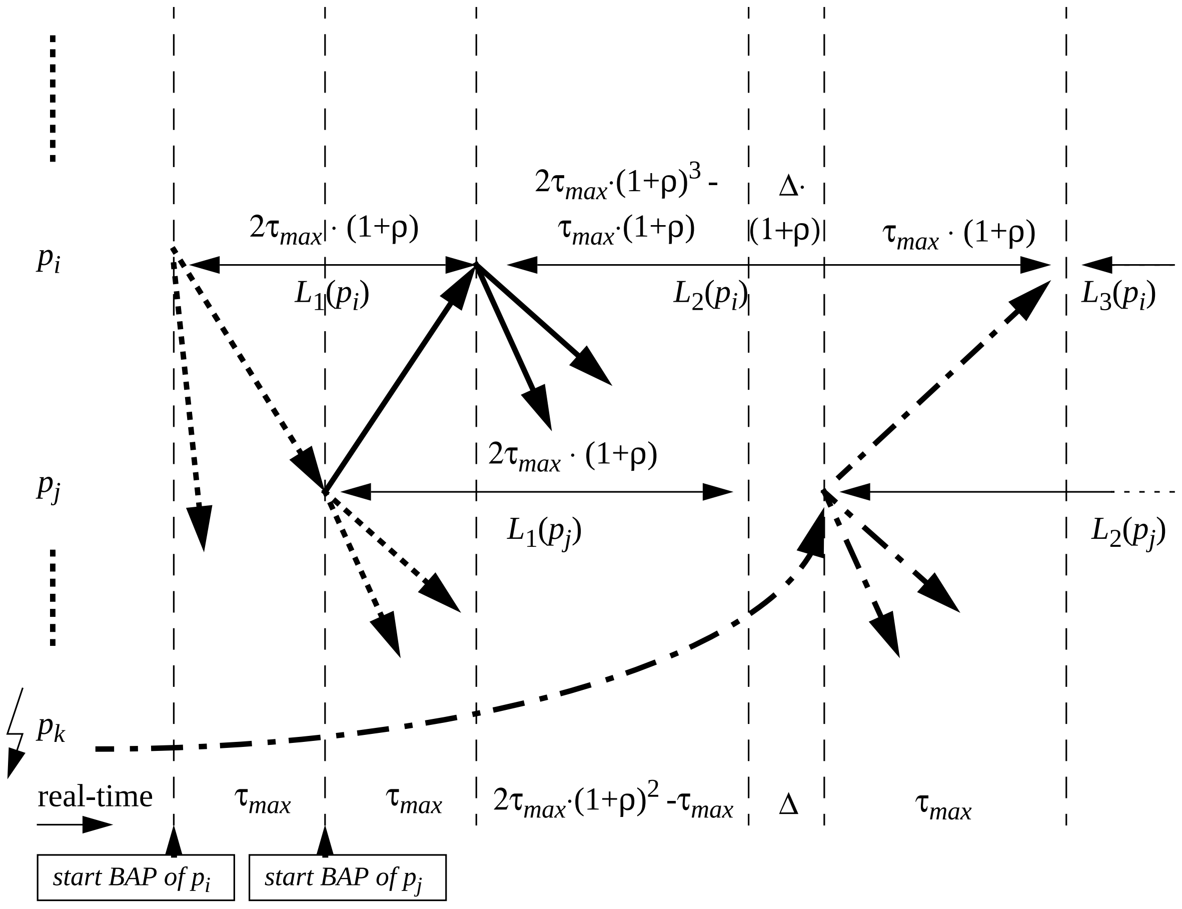
$$L_2 = 2\tau_{max} \cdot (1+\rho)^3 + \Delta \cdot (1+\rho)$$

$$(\forall\, h \in [3,T+1]{::}\ L_h = [L_{h-1} \cdot (1+\rho) + \Delta] \cdot (1+\rho))$$

A3.16. A processor may concurrently execute different ICAs. We assume that messages of different ICAs can be distinguished, such that every correct processor can determine when it receives the first valid message of a new ICA.

Now, in a similar way as it is done in Section 3.3.9. for authenticated self-synchronizing BAPs, it will be proved that, provided that the unforgeability properties UP1 and UP2 and the assumptions A3.1 through A3.3, A3.5 through A3.8, and A3.15 and A3.16 hold, the authenticated self-synchronizing ICA satisfies the conditions ICA1 and ICA2 stated in Section 3.1.2.

In Figure 3.14., as an example, we indicate how protocol synchronization between two correct processors p_i and p_j (p_i, $p_j \in N$) is achieved in an authenticated self-synchro-


p_i
p_j
p_k
$2\tau_{max} \cdot (1+\rho)$
$L_1(p_i)$
$2\tau_{max} \cdot (1+\rho)^3 -$
$\tau_{max} \cdot (1+\rho)$
$L_2(p_i)$
$\Delta \cdot$
$(1+\rho)$
$\tau_{max} \cdot (1+\rho)$
$L_3(p_i)$
$2\tau_{max} \cdot (1+\rho)$
$L_1(p_j)$
$L_2(p_j)$
real-time
τ_{max}
τ_{max}
$2\tau_{max} \cdot (1+\rho)^2 - \tau_{max}$
Δ
τ_{max}
start BAP of p_i
start BAP of p_j

$\rho \geq 0 \wedge \Delta \geq 0$

$(\forall\, p \in N : \forall\, k \in [1,T+1] :: correct(p) \Rightarrow (1 / (1+\rho)) \cdot L_k(p) \leq L_k \leq (1+\rho) \cdot L_k(p))$

$L_1 = 2\tau_{max} \cdot (1+\rho)$

$L_2 = 2\tau_{max} \cdot (1+\rho)^3 + \Delta \cdot (1+\rho)$

$(\forall\, k \in [3,T+1] :: L_k = [L_{k-1} \cdot (1+\rho) + \Delta] \cdot (1+\rho))$

For processor p_i and p_j it holds that

$(\forall\, k \in [1,T+1] :: (1/(1+\rho)) \cdot L_k(p_i) = (1+\rho) \cdot L_k(p_j) = L_k)$

Figure 3.14. Protocol synchronization between correct processors p_i and p_j in an authenticated self-synchronizing ICA, with faulty processor p_k and $\rho \geq 0$ and $\Delta \geq 0$.

nizing ICA, in the presence of a faulty processor p_k ($p_k \in N$). Proofs for the general case are given by the lemma's and theorems in this section. The time needed to communicate messages, the speeds at which the processor clocks of different processor operate, and the clock value measurement uncertainties have been chosen such that the difference in protocol synchronization between p_i and p_j is maximal. The length of the first communication phase is chosen such that every valid message that is sent directly from a source to some correct processor p_x ($p_x \in N$), and which belongs to a BAP (belonging to the ICA) which may have been started in time, is accepted by p_x in communication phase 1 of its sub-ICA. In this way, the valid message sent directly from faulty processor p_k to p_j is received in time and accepted by p_j in communication phase 1 of p_j's sub-ICA. Hence, this message must have been received and accepted by processor p_i before the end of communication phase 2 of p_i's sub-ICA. See Figure 3.14.

THEOREM 3.11.
Any valid message generated by a correct source *s* in a BAP belonging to an ICA, which is sent directly from *s* to a correct processor *c*, is timely received and accepted by *c*.

Proof:
Assume a correct source *s* of a certain BAP belonging to the ICA generates and sends a valid message *m* directly to a correct processor *c*. Then, we must prove that *c* timely receives and accepts *m*.

We distinguish between two cases and show that, in both cases, *c* accepts *m*.

Case 1. Processor c has already started the first communication phase of its sub-ICA at receipt of message m.
Since processor *c* is a correct processor, at the start of the first communication phase of its sub-ICA, *c* has generated and sent a valid message directly to every processor in $N \setminus \{c\}$. As a consequence, after a real-time interval of at most τ_{max} after *c* has started its sub-ICA, every processor in $N \setminus \{c\}$ (thus also source *s*) has received a valid message from *c*. Hence, if source *s* had not yet started its sub-ICA, *s* is now triggered to start its sub-ICA at receipt of the message from *c*. Since we assumed that source *s* is correct and generates and sends a valid message directly to processor *c*, this message will arrive in *c* after a real-time interval of at most τ_{max} after *s* has sent it (which is at or before receipt of the message from *c*). So *c* receives a valid message from *s* after a real-time interval of at most $2\tau_{max}$ after *c* has started the first communication phase of its sub-ICA. Since, by A3.2 and A3.15, the first communication phase of *c*'s sub-ICA has a real-time length of at least $2\tau_{max}$, the message from *s* sent to *c* arrives in *c* in the first communication phase of *c*'s sub-ICA, and hence, this message is accepted by *c*.

Case 2. Processor c starts the first communication phase of its sub-ICA at receipt of message m.
In this case, the message received from *s* is received in the first communication phase of *c*'s sub-ICA. Since the message was sent directly from *s* to *c*, its path information contains only 2 processors, and hence, the message is accepted.

Thus, in both cases, *c* accepts *m*. This proves Theorem 3.11. ❏

LEMMA 3.12.1.
Once a correct processor *p* has concluded the first communication phase of its sub-ICA, then, within a real-time interval of $2\tau_{max}\cdot(1+\rho)^2 - \tau_{max} + \Delta$, all other correct processors have also concluded the first communication phase of their sub-ICA.

Proof:
The proof is similar to that of Lemma 3.6.1. for authenticated self-synchronizing BAPs.

Notice that the maximum real-time difference between the end of the first communication phase of the sub-ICAs of two different correct processors *p* and *q* is determined by the maximum real-time difference between the start of the first communication phase of the sub-ICA of processor *p* and *q*, and by the maximum real-time difference in the length of the first communication phase in the sub-ICAs of processor *p* respectively, of processor *q*.

The real-time length of any communication phase *i* ($1 \leq i \leq T+1$) of the sub-ICA of any correct processor *c* is ***maximal***, if *c*'s processor clock runs a factor (1+ρ) slower than real-time, and *c* detects the end of communication phase *i* only after a real-time interval of length Δ after its processor clock indicates that communication phase *i* is over. By A3.15, for any correct processor *c*, the length L_1 of the first communication phase of *c*'s sub-ICA, as measured by *c*'s processor clock, is equal to $2\tau_{max}\cdot(1+\rho)$. Hence, the *maximum* real-time length of the first communication phase of the sub-ICA of any correct processor *c* is $(1+\rho)\cdot L_1 + \Delta$, which is equal to $2\tau_{max}\cdot(1+\rho)^2 + \Delta$.

The real-time length of any communication phase *i* ($1 \leq i \leq T+1$) of the sub-ICA of any correct processor *c* is ***minimal***, if *c*'s processor clock runs a factor (1+ρ) faster than real-time, and *c* immediately detects the end of communication phase *i*, as soon as its processor clock indicates that communication phase *i* is over. Hence, the *minimum* real-time length of the first communication phase of the sub-ICA of any correct processor *c* is $L_1 / (1+\rho)$, which is equal to $2\tau_{max}$.

The real-time difference between the start of the first communication phase of the sub-ICA of two different correct processors is ***maximal***, if one of the correct processors, say *p*, is the first correct processor[49] that starts the first communication phase of its sub-ICA, and the other correct processor, say *q*, only starts the first communication phase of its sub-ICA at receipt of the first valid message from processor *p*, and the maximum real-time interval τ_{max} is needed to communicate this message from *p* to *q*. Hence, the maximum real-time difference between the start of the first communication phase of the sub-ICA of two correct processors is equal to τ_{max}. This can be seen as follows.

The above situation may occur, if processor *p* initiates the ICA, or if the ICA is initiated by a set of colluding faulty processors, and processor *p* is the first correct processor that receives a valid message from the ICA.

If processor *p* initiates the ICA, processor *p* will broadcast a message to all other proc-

49. Notice that ***more than one*** correct processor may be the first to start the first communication phase of its sub-ICA. Viz., the ICA may be initiated by a set of colluding faulty processors, and they may cause different valid messages to simultaneously arrive in more than one correct processor. Each of these correct processors will be triggered to start the first communication phase of its sub-ICA at receipt of this message, so each of these correct processors is the first to start the first communication phase of its sub-ICA.

essors in the system, and start the first communication phase of its sub-ICA. The message sent from processor *p* to processor *q* will arrive after a real-time interval of τ_{max}, and cause processor *q* to start the first communication phase of its sub-ICA at receipt of this message, if this message is the first valid message that *q* receives, i.e., if processor *q* has not previously received a valid message from another processor *r*, which has relayed to *q* the message received from *p*.

If processor *p* is the first correct processor that receives a valid message from an ICA initiated by a set of colluding faulty processors, then at receipt of this message, processor *p* will start the first communication phase of its sub-ICA, and relay the received message *m* to all processors, the identification of which is not in the path described by the path information of *m*. The message is also relayed to processor *q*, since the identification of processor *q* cannot be present in the path described in the path information of *m*, since unforgeability property UP1 in Section 3.3.3. implies that, in this case, *q* must have received and accepted *m*, and hence, *q* must have received a valid message from the ICA before *p* receives it. This contradicts our assumption that processor *p* was the first correct processor that receives a valid message from the ICA.

The message sent from processor *p* to processor *q* will arrive after a real-time interval of τ_{max}, and cause processor *q* to start the first communication phase of its sub-ICA at receipt of this message, if this message is the first valid message that *q* receives, i.e., if processor *q* has not previously received a valid message from another processor *r*.

The real-time difference between the end of the first communication phase of the sub-ICAs of two different correct processors is ***maximal***, if the real-time difference between the start of the first communication phase is maximal, and the real-time difference in lengths of the first communication phase of the sub-ICAs in the different processors is maximal, i.e., if the length of the first communication phase of the sub-ICA of the correct processor that starts its sub-ICA first is minimal, whereas the length of the first communication phase of the sub-ICA of the correct processor that starts its sub-ICA τ_{max} later is maximal.

Let *p* be the first correct processor that starts its sub-ICA, say at time $t_{earliest\_start}$. Then, the earliest time at which processor *p* can conclude the first communication phase of its sub-ICA is at time $t_{earliest\_end} = t_{earliest\_start} + 2\tau_{max}$ (i.e., if the real-time length of the first communication phase of *p*’s sub-ICA is minimal). Then, the latest moment at which any other correct processor *q* can start the first communication phase of its sub-ICA, is at real-time $t_{latest\_start} = t_{earliest\_start} + \tau_{max}$ (i.e., in case *q* starts its sub-ICA as a result of receiving the first valid message of the ICA from *p*, τ_{max} after *p* sent it). The latest moment at which *q* can conclude the first communication phase of its sub-ICA is at $t_{latest\_end} = t_{latest\_start} + 2\tau_{max}\cdot(1+\rho)^2 + \Delta$ (i.e., in case the real-time length of the first communication phase of *q*’s sub-ICA is maximal).

Hence, as soon as a correct processor has concluded the first communication phase of its sub-ICA, after a real-time interval of at most

$$t_{latest\_end} - t_{earliest\_end} =$$

$$(t_{earliest\_start} + \tau_{max} + 2\tau_{max}\cdot(1+\rho)^2 + \Delta) - (t_{earliest\_start} + 2\tau_{max}) =$$

$$2\tau_{max} \cdot (1+\rho)^2 - \tau_{max} + \Delta,$$

all other correct processors have also concluded the first communication phase of their sub-ICA. This concludes Lemma 3.12.1. ❑

LEMMA 3.12.2.
After communication phase i ($1 \le i \le T$), in real-time, the sub-ICAs of all correct processors are synchronized within a real-time bound of $(L_{i+1} / (1+\rho)) - \tau_{max}$.

Proof:
The proof is similar to that of Lemma 3.6.2. for authenticated self-synchronizing BAPs. Whereas the lengths of the various communication phases of the sub-BAP executed in any correct processor, as measured by the processor clock of this correct processor is given by A3.9, in the proof below, we will use the lengths of the various communication phases of the sub-ICA executed in any correct processor, as measured by the processor clock of this correct processor, as indicated in A3.15.

We will prove this lemma by induction on i.

Basis: i = 1.
From Lemma 3.12.1. we know that after communication phase 1, the sub-ICAs of all correct processors have been synchronized within a real-time bound of $2\tau_{max} \cdot (1+\rho)^2 - \tau_{max} + \Delta$, which is equal to $(L_2 / (1+\rho)) - \tau_{max}$.

Induction step: i > 1.
We assume that Lemma 3.12.2. holds for $j = i - 1$. Thus, the induction hypothesis is:
After communication phase i - 1(2 ≤ i ≤ T), in real-time, the sub-ICAs of all correct processors are synchronized within a real-time bound of $L_i / (1+\rho) - \tau_{max}$.

Now, we prove that Lemma 3.12.2. also holds for $j = i$.

Notice that the maximal real-time difference in synchronization occurs between the sub-ICA of a correct processor s which runs a factor $(1+\rho)$ slower than real-time and that of a correct processor f which runs a factor $(1+\rho)$ faster than real-time. The real-time difference is maximal if processor f immediately detects the end of its communication phase, whereas processor s only detects it after real-time Δ.

For all i (with $1 \le i \le T+1$) and any correct processor p, let $L_i(p)$ be the real-time length of communication phase i in p. For a correct processor s that runs a factor $(1+\rho)$ slower than real-time, the maximum real-time length of communication phase i in s is $L_i(s) = L_i \cdot (1+\rho) + \Delta$. For a correct processor f that runs a factor $(1+\rho)$ faster than real-time, the minimum real-time length of communication phase i in f is $L_i(f) = L_i / (1+\rho)$. So, the real-time difference in protocol synchronization increases by an amount less than or equal to $L_i(s) - L_i(f)$. From the induction hypothesis, we know that the real-time difference after communication phase $i - 1$ is less than or equal to $L_i / (1+\rho) - \tau_{max}$, which is equal to $L_i(f) - \tau_{max}$. After communication phase i, the maximal real-time difference in protocol synchronization is $(L_i(f) - \tau_{max}) + (L_i(s) - L_i(f)) = L_i(s) - \tau_{max}$, which is equal to $L_i \cdot (1+\rho) + \Delta - \tau_{max}$. Since, by A3.15, for $i > 1$, $L_{i+1} = [L_i \cdot$

$(1+\rho) + \Delta]\cdot (1+\rho)$, the maximal real-time difference in protocol synchronization after communication phase i is $L_i \cdot (1+\rho) + \Delta - \tau_{max} = L_{i+1} / (1+\rho) - \tau_{max}$. This proves Lemma 3.12.2. ❑

LEMMA 3.12.3.
Let $m(mv,(\underline{\mathbf{p}}))$ be any valid message which is timely received and accepted by a correct processor c in communication phase i ($1 \leq i \leq T$) of c's sub-ICA and assume that c relays $m(mv,(\underline{\mathbf{p}}))$ to a correct processor d, i.e., processor c sends $m(mv,(\underline{\mathbf{p}};d))$ to d. Then, message $m(mv,(\underline{\mathbf{p}};d))$ arrives in processor d in or before communication phase i+1 of d's sub-ICA.

Proof:
Identical to the proof of Lemma 3.6.3. for authenticated self-synchronizing BAPs, except for the fact that processor c and d execute a sub-ICA instead of a sub-BAP, and the lengths of the communication phases of the sub-ICA executed on a correct processor, as measured by the processor clock of that correct processor, are given by A3.15 (whereas, in Lemma 3.6.3., the lengths of the various communication phases of the sub-BAP executed on a correct processor, as measured by the processor clock of that correct processor are given by A3.9). ❑

THEOREM 3.12.
If a correct processor accepts a valid message of a BAP belonging to the ICA with a certain message value *mv*, then all correct processors will accept a valid message of this BAP with message value *mv* before the end of the ICA.

Proof:
Identical to the proof of Theorem 3.6. for authenticated self-synchronizing BAPs, except for the fact that the correct processors execute an ICA instead of a BAP. ❑

With the help of Theorem 3.11. and 3.12., it is possible to prove that the authenticated self-synchronizing ICA presented in Section 3.4.1. satisfies conditions ICA1 and ICA2, which have been stated in Section 3.1.2. as follows:

ICA1. All correct processors agree on the initial value they think they have received from each of the processors.
ICA2. For every processor p, it holds that if p is correct, the above-mentioned agreement equals the initial value actually sent by processor p.

The proof is given by the following theorem.

THEOREM 3.13.
Provided that the unforgeability properties UP1 and UP2 from Section 3.3.3. and assumptions A3.1. through A3.3., and A3.5 through A3.8. from Section 3.3.8. and A3.15 and A3.16 in this section hold, the authenticated self-synchronizing ICA presented in this section satisfies conditions ICA1 and ICA2.

Proof:
We first prove that the authenticated self-synchronizing ICA satisfies ICA1.

As stated before, an authenticated self-synchronizing ICA runs on a system, *N*, of *N* processors, and consists of execution of *N* authenticated self-synchronizing BAPs, one per processor.

From Theorem 3.12., we know that if a correct processor accepts a valid message of a BAP belonging to the ICA with a certain message value *mv*, then all correct processors will accept a valid message of this BAP with message value *mv* before the end of the ICA. Hence, for every BAP belonging to the ICA, the sets of message values of valid messages of the BAP accepted during the ICA are equal for all correct processors.

The decision taken in a correct processor about the initial value sent by the source of a certain BAP belonging to the ICA is based on applying a decision function on the set of message values of the valid messages of the BAP accepted during the multicast process of the ICA. Since, for every BAP belonging to the ICA, all correct processors apply the same decision function on the same set of message values, the decisions of all correct processors are the same for every BAP belonging to the ICA, and thus, condition ICA1 is satisfied.

We now prove that the authenticated self-synchronizing ICA satisfies ICA2.

We prove that in an arbitrary BAP belonging to the ICA, if the source *s* of the BAP maintains an initial value *mv*, and the source is correct, then the decision about this BAP belonging to the ICA in any correct processor will be equal to *mv*.

We assume that the source *s* of a certain BAP belonging to the ICA is correct. This implies that the source initiates the BAP belonging to the ICA by sending a valid message *m*(*mv*,(*s;x*)) to any other processor *x* in the system. Furthermore, it accepts a copy of the message itself in order to use it in the decision-making process of the BAP belonging to the ICA. From Theorem 3.11., we know that, if the source *s* which initiates the BAP belonging to the ICA is *correct*, any valid message sent directly from the source *s* to a correct processor *c* is timely received and accepted in processor *c*. Hence, any correct processor *c* will accept a valid message *m*(*mv*,(*s;c*)) received from the source *s*. For any correct processor *c*, the set of message values on which *c* will base its decision of the BAP belonging to the ICA will thus at least contain message value *mv* of the message received from the source. However, the set of message values on which *c* bases its decision cannot contain other message values *mv*' ≠ *mv*, because the source *s* is correct and hence, only generates and multicasts valid messages with message value *mv*, and, by the unforgeability properties UP1 and UP2, faulty processors are not able to generate and multicast valid messages *m*(*mv*',(*s;c*)) with *mv*' ≠ *mv*, since such messages need the signature of processor *s*, which the faulty processors are unable to produce, since *s* is correct. Hence, any correct processor (including the source itself, since it has accepted a copy of its own message) will decide *mv*.

Since we have investigated the decision-making process in an arbitrary BAP belonging to the ICA, we may conclude that the above holds for any BAP belonging to the ICA. Hence, condition ICA2 is satisfied.

This concludes the proof of Theorem 3.13. ❏

3.4.2. A description of an optimized authenticated self-synchronizing ICA

An optimized authenticated self-synchronizing ICA has similarities with the optimized authenticated self-synchronizing BAP given in Section 3.3.7. In fact, an optimized ICA runs on a system, N, of N processors, and consists of execution of N optimized BAPs. Every processor $p \in N$ is the source in one of the BAPs belonging to the ICA. Of these BAPs, for any processor $p \in N$, we will denote the BAP in which processor p acts as the source as BAP_p.

The remarks that were made in Section 3.4.1. with regard to authenticated self-synchronizing ICAs are also applicable to optimized authenticated self-synchronizing ICAs, with that difference that, by A3.17., for all k, with $1 \le k \le T{+}1$, L_k is recursively defined by

$$L_1 = (T+1)\cdot\tau_{max}\cdot(1+\rho)$$
$$L_2 = (T+1)\cdot\tau_{max}\cdot(1+\rho)^3 - (T-1)\cdot\tau_{max}\cdot(1+\rho) + \Delta\cdot(1+\rho)$$
$$(\forall\, h \in [3,T+1]\colon L_h = [L_{h-1}\cdot(1+\rho)+\Delta]\cdot(1+\rho))$$

An optimized authenticated self-synchronizing ICA can be described as follows:
Initially, for each processor $i \in N$, for all j, with $1 \le j \le T+1$, $endcommphase_i(j) = \infty$

For some (possibly more than one) processor $s \in N$:

Event: *Processor s is triggered to start the ICA at time $C(s,t)$.*

Action: *Processor s acts as the source in BAP_s belonging to the ICA and generates and sends a message $m(mv,(s;k))$ to every processor $k \in S_{(s)}$. The source accepts a copy of its own message in order to use it in the decision-making process of BAP_s.*

Simultaneously, the source starts the first communication phase of its sub-ICA. Furthermore, the source calculates at which clock values $C(s,t')$ of its processor clock each of the $T+1$ communication phases of its sub-ICA ends. For all h, with $1 \le h \le T+1$, it holds that

$$endcommphase_s(h) = C(s,t) + \sum_{k=1}^{h} L_k$$

For each processor $i \in N$:

Event: *$C(i,t) \ge endcommphase_i(T+1)$*

Action: *processor i decides the outcome of the ICA on basis of the decisions from all of the BAP_k (for any $k \in N$) it has calculated.*

Event: *message $mess_j(i)$ (with $1 \le j \le M_i$) received at time $C(i,t)$*

Action: *1. Check if $mess_j(i)$ is a valid message, i.e., check if $mess_j(i) \in ValidMessages$.*

If $invalid(mess_j(i))$ then reject $mess_j(i)$ and abort, i.e., do no perform further actions on $mess_j(i)$.

If $valid(mess_j(i))$, then $mess_j(i) = m(mv,(\underline{\mathbf{q}}))$ of some BAP_s belonging to the ICA (with $s = first((\underline{\mathbf{p}};i))$, for some path $(\underline{\mathbf{q}}) \in ValidPaths$ and *some mv ∈ MessageValues. If last((**q**)) ≠ i, then reject* $mess_j(i)$ *and abort, otherwise* $mess_j(i)$ *= m(mv,(**p**;i)) for path (**p**) ∈ ValidPaths with (**p**) = prefix((**q**)).*

2. *Check if message m(mv,(**p**;i)) was expected according to i's subset selection rules, i.e., check if i ∈* $S_{(\mathbf{p})}$.
   If i ∉ $S_{(\mathbf{p})}$, *then reject m(mv,(**p**;i)) and abort.*
3. *Check if a valid message* $mess_k(i)$ *(with 1 ≤ k < j) with path(*$mess_k(i)$*) = (**p**;i) has already been accepted.*
   If such a message has already been accepted, then reject $mess_j(i)$ *and abort.*
4. *Check if m(mv,(**p**;i)) has been received in time, i.e., check if C(i,t) ≤* $endcommphase_i$*(length((**p**;i))-1). If m(mv,(**p**;i)) has been received in time, i accepts m(mv,(**p**;i)), otherwise i rejects m(mv,(**p**;i)).*
5. *If m(mv,(**p**;i)) is the first valid message of the ICA that i receives (and hence, accepts), i starts the first communication phase of its sub-ICA at receipt of m(mv,(**p**;i)). Processor i now calculates at which clock values C(i,t') of its processor clock each of the T+1 communication phases of its sub-ICA ends. For all h, with 1 ≤ h ≤ T+1, it holds that*

   $$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

   Furthermore, processor i is triggered to start its own BAP (BAP_i*) belonging to the ICA. Hence, processor i acts as the source in* BAP_i *belonging to the ICA and generates and sends a message m(mv,(i;k)) to every processor k ∈* $S_{(i)}$. *Processor i accepts a copy of its own message in order to use it in the decision-making process of* BAP_i.
5. *If i accepted m(mv,(**p**;i)) and path(m(mv,(**p**;i))) contains fewer than T+2 processors, for every processor h ∈* $S_{(\mathbf{p};i)}$, *i signs message m(mv,(**p**;i)) and i relays m(mv,(**p**;i)) to h, immediately after m(mv,(**p**;i)) was received by i. As stated in Section 3.3.3.3., relaying message m(mv,(**p**;i)) to any processor h results in m(mv,(**p**;i;h)) being sent to h.*

All correct processors are assumed to execute the above-described algorithm. The behaviour of faulty processors, however, may deviate arbitrarily from the described algorithm.

Assume an optimized authenticated self-synchronizing ICA is executed in a fully-connected synchronous system *N* consisting of *N* processors (including a source *s*), up to *T* of which may behave maliciously. We assume that for any optimized authenticated self-synchronizing BAP belonging to the ICA, the unforgeability properties UP1 and UP2 hold, and assumptions A3.1 through A3.3, A3.5 and A3.6, A3.10 through A3.13 and A3.16 and A3.17 are satisfied, where A3.17 is given by

A3.17. Every processor in the system may have a different view of time and of the start and end of communication phases in the ICA. For all *i*, with 1 ≤ *i* ≤ *T*+1, for any correct processor *p* ∈ *N*, let L_i be the length of communication phase i of p's sub-ICA, as viewed from p. Then:

$$L_1 = (T+1) \cdot \tau_{max} \cdot (1+\rho)$$

$$L_2 = (T+1) \cdot \tau_{max} \cdot (1+\rho)^3 + \Delta \cdot (1+\rho)$$

$$(\forall\ h \in [3,T+1]: L_h = [L_{h-1} \cdot (1+\rho)+\Delta] \cdot (1+\rho))$$

Now, in a similar way as it is done in Section 3.3.10. for optimized authenticated self-synchronizing BAPs, it will be proved that, provided that the unforgeability properties UP1 and UP2, and assumptions A3.1 through A3.3, A3.5 and A3.6, A3.10 through A3.13 and A3.16 and A3.17 hold, the optimized authenticated self-synchronizing ICA satisfies the conditions ICA1 and ICA2 stated in Section 3.1.2.

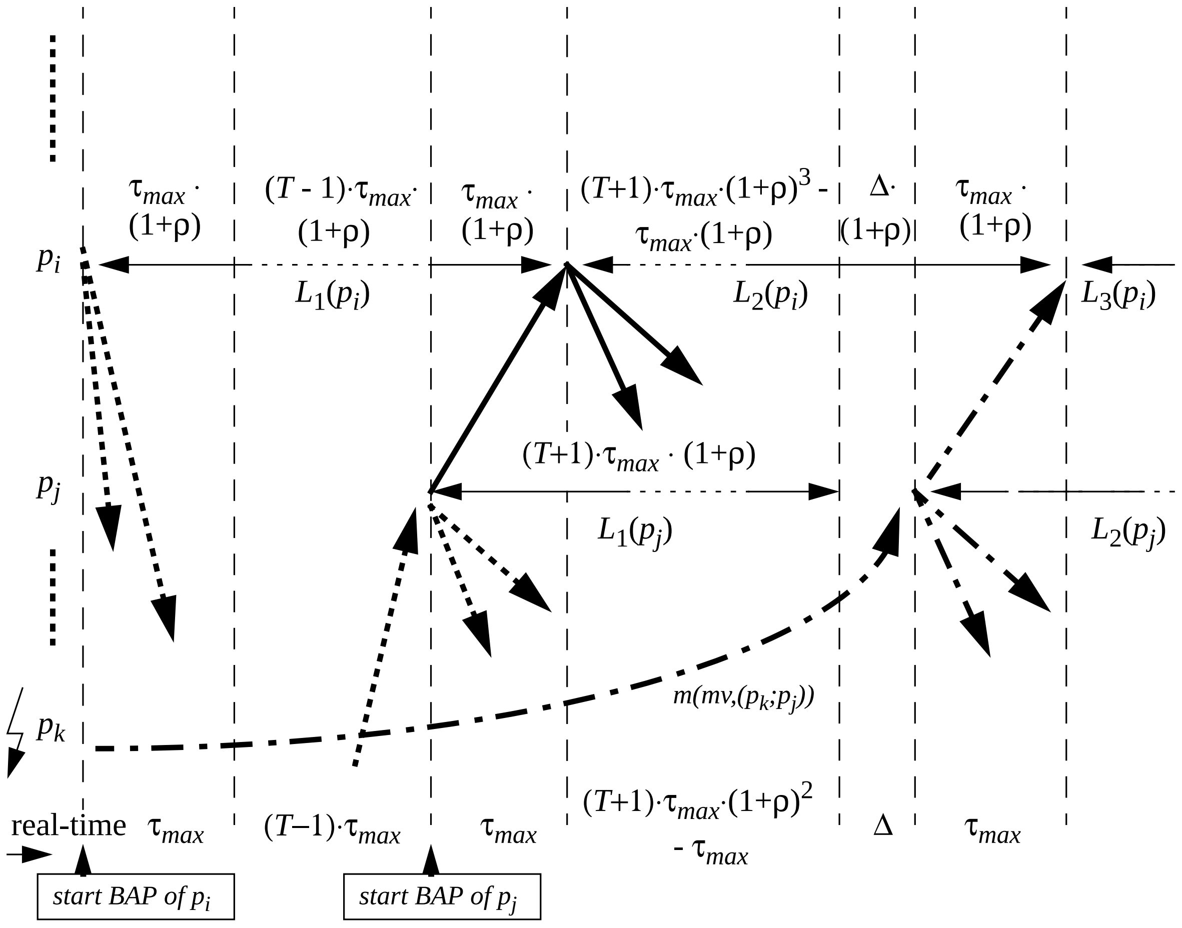
As an example, in Figure 3.15., we indicate how protocol synchronization between two correct processors p_i and p_j (p_i, $p_j \in N$) is achieved in an optimized authenticated self-synchronizing ICA, in the presence of a faulty processor p_k ($p_k \in N$). Proofs for the general case are given by the lemma's and theorems in this section. The time needed to communicate messages, the speeds at which the processor clocks of different processors operate, and the clock value measurement uncertainties have been chosen such that the difference in protocol synchronization between p_i and p_j is maximal. By analogy with the optimized authenticated self-synchronizing BAPs described in Section 3.3.7., the difference in protocol synchronization is maximal, if processor p_j expects that the first valid message it receives has path information describing a path with length greater than or equal to 2. In this case, the valid message $m(mv, (p_k;p_j))$, sent directly from faulty processor p_k to processor p_j may be expected, and hence, be accepted by processor p_j during communication phase 1 of its sub-ICA, after which processor p_j relays the message to p_i. As a consequence, processor p_i should receive and accept the message before the end of communication phase 2 of p_i's sub-ICA.

THEOREM 3.14.
Any valid message generated by a correct source s in a BAP belonging to an ICA, which is sent within T relay steps from s to a correct processor c, is timely received and accepted by c.

Proof:
Assume a correct source s of a certain BAP belonging to the ICA generates and sends a valid message m within T relay steps to a correct processor c. Then, we must prove that c accepts m.

We distinguish between two cases and show that, in both cases, c accepts m.

Case 1. Processor c has already started the first communication phase of its sub-ICA at receipt of message m.
Since processor c is a correct processor, at the start of the first communication phase of its sub-ICA, c has generated and sent a valid message directly to every processor in $S_{(c)}$, according to the subset selection rules for that message (by A3.10). By A3.12, these subset selection rules are such that every valid message from source c is relayed to all correct processors within T relay steps. As a consequence, after a real-time interval of at most $T \cdot \tau_{max}$ after c has started its sub-ICA, every processor in $N \setminus \{c\}$ (thus


$\tau_{max}\cdot(1+\rho)$
$(T-1)\cdot\tau_{max}\cdot(1+\rho)$
$\tau_{max}\cdot(1+\rho)$
$(T+1)\cdot\tau_{max}\cdot(1+\rho)^3 - \tau_{max}\cdot(1+\rho)$
$\Delta\cdot(1+\rho)$
$\tau_{max}\cdot(1+\rho)$
p_i
$L_1(p_i)$
$L_2(p_i)$
$L_3(p_i)$
$(T+1)\cdot\tau_{max}\cdot(1+\rho)$
p_j
$L_1(p_j)$
$L_2(p_j)$
$m(mv,(p_k;p_j))$
p_k
real-time
τ_{max}
$(T-1)\cdot\tau_{max}$
τ_{max}
$(T+1)\cdot\tau_{max}\cdot(1+\rho)^2 - \tau_{max}$
Δ
τ_{max}
start BAP of p_i
start BAP of p_j

$\rho \geq 0 \wedge \Delta \geq 0$

$(\forall\, p \in N : \forall\, k \in [1,T+1] :: correct(p) \Rightarrow (1 / (1+\rho))\cdot L_k(p) \leq L_k \leq (1+\rho)\cdot L_k(p))$

$L_1 = (T+1)\cdot\tau_{max}\cdot(1+\rho)$
$L_2 = (T+1)\cdot\tau_{max}\cdot(1+\rho)^3 + \Delta\cdot(1+\rho)$
$(\forall\, k \in [3,T+1] :: L_k = [L_{k-1}\cdot(1+\rho) + \Delta]\cdot(1+\rho))$

For processor p_i and p_j it holds that
$(\forall\, k \in [1,T+1] :: (1/(1+\rho))\cdot L_k(p_i) = (1+\rho)\cdot L_k(p_j) = L_k)$

Processor p_j expects that the first valid message it receives has path information describing a path with length ≥ 2.

Figure 3.15. *Protocol synchronization between correct processors p_i and p_j in an optimized authenticated self-synchronizing ICA, with faulty processor p_k and $\rho \geq 0$ and $\Delta \geq 0$.*

also source s) has received a valid message from c. Hence, if source s had not yet started its sub-ICA, s is now triggered to start its sub-ICA at receipt of the message from c. Since we assumed that source s generates and sends a valid message directly and in time to processor c, this message will arrive in c after a real-time interval of at most τ_{max} after s has sent it (which is at or before receipt of the message from c). So c receives a valid message from s after a real-time interval of at most $(T+1)\cdot\tau_{max}$ after c has started the first communication phase of its sub-ICA. Since, by A3.2 and A3.17, the first communication phase of c's sub-ICA has a real-time length of at least $(T+1)\cdot\tau_{max}$, the message from s sent to c arrives in c in the first communication phase of c's sub-ICA, and hence, this message is accepted by c.

Case 2. Processor c starts the first communication phase of its sub-ICA at receipt of message m.

In this case, the message received from s is received in the first communication phase of c's sub-ICA. Since the message was sent within T relay steps from s to c, its path information contains at least 2 processors, and hence, the message is accepted.

Thus, in both cases, c accepts m. This proves Theorem 3.14. ❑

LEMMA 3.15.1.

Once a correct processor p has concluded the first communication phase of its sub-ICA, then, within a real-time interval of $(T+1)\cdot\tau_{max}\cdot(1+\rho)^2 - \tau_{max} + \Delta$, all other correct processors have also concluded the first communication phase of their sub-ICA.

Proof:

Identical to the proof of Lemma 3.12.1., except for the fact that the maximum real-time difference between the start of the first communication phase of the sub-ICA of two different correct processors is not equal to τ_{max} (as in Lemma 3.12.1) but equal to $T\cdot\tau_{max}$, and the length of the first communication phase of the sub-ICA of a correct processor, as measured by the processor clock of that correct processor is not equal to $2\tau_{max}\cdot(1+\rho)$ (as in Lemma 3.12.1) but equal to $(T+1)\cdot\tau_{max}\cdot(1+\rho)$.

In an optimized authenticated self-synchronizing ICA, the above maximum real-time difference between the start of the first communication phase of the sub-ICA of two different correct processors occurs, if one of the correct processors, say p, initiates the ICA, and the subset selection rules of all BAPs belonging to the ICA are chosen such that the other correct processor, say q, only receives the first valid message from the ICA after T relay steps, and, in each relay step, a real-time interval of maximal length τ_{max} is needed to relay the message from the sender to the receiver of the message. While processor p starts the first communication phase of its sub-ICA as soon as it has initiated the ICA, processor q can only start the first communication phase of its sub-ICA a real-time period of $T\cdot\tau_{max}$ after p has initiated the BAP (i.e., as soon as processor q has received the first valid message of the ICA).

By A3.17, for any correct processor c, the length L_1 of the first communication phase of c's sub-ICA, as measured by c's processor clock, is equal to $(T+1)\cdot\tau_{max}\cdot(1+\rho)$. Hence, the maximum real-time length of the first communication phase of the sub-ICA of any correct processor c is $(1+\rho)\cdot L_1 + \Delta$, which is equal to $(T+1)\cdot\tau_{max}\cdot(1+\rho)^2 + \Delta$, whereas the minimum real-time length of the first communication phase of the sub-ICA of any correct processor is L_1 / $(1+\rho)$, which is equal to $(T+1)\cdot\tau_{max}$.

As in Lemma 3.12.1., the maximal real-time difference between the end of the first communication phase of the sub-ICAs of two different correct processors p and q is determined by the maximum real-time difference between the start of the first communication phase of the sub-BAP of processor p and q, and by the maximum real-time difference in the length of the first communication phase in the sub-BAPs of processor p respectively, of processor q.

In a similar way as it is done in Lemma 3.12.1., we will now calculate the maximal real-time difference between the end of the first communication phase of the sub-ICAs of two different correct processors.

The real-time difference between the end of the first communication phase of the sub-ICAs of two different correct processors is ***maximal***, if the real-time difference between the start of the first communication phase is maximal, and the real-time difference in lengths of the first communication phase of the sub-ICAs in the different processors is maximal, i.e., if the length of the first communication phase of the sub-ICA of the correct processor that starts its sub-ICA first is minimal, whereas the length of the first communication phase of the sub-ICA of the correct processor that starts its sub-ICA $T\cdot\tau_{max}$ later is maximal.

Let p be the first correct processor that starts its sub-ICA, say at time $t_{earliest\_start}$. Then, the earliest time at which processor p can conclude the first communication phase of its sub-ICA is at time $t_{earliest\_end} = t_{earliest\_start} + (T+1)\cdot\tau_{max}$ (i.e., if the real-time length of the first communication phase of p's sub-ICA is minimal). Then, the latest moment at which any other correct processor q can start the first communication phase of its sub-ICA, is at real-time $t_{latest\_start} = t_{earliest\_start} + T\cdot\tau_{max}$ (i.e., in case q starts its sub-ICA as a result of receiving the first valid message of the ICA from p, $T\cdot\tau_{max}$ after p sent it). The latest moment at which q can conclude the first communication phase of its sub-ICA is at $t_{latest\_end} = t_{latest\_start} + (T+1)\cdot\tau_{max}\cdot(1+\rho)^2 + \Delta$ (i.e., in case the real-time length of the first communication phase of q's sub-ICA is maximal).

Hence, as soon as a correct processor has concluded the first communication phase of its sub-ICA, after a real-time interval of at most

$$t_{latest\_end} - t_{earliest\_end} =$$
$$(t_{earliest\_start} + T\cdot\tau_{max} + (T+1)\cdot\tau_{max}\cdot(1+\rho)^2 + \Delta) - (t_{earliest\_start} + (T+1)\cdot\tau_{max}) =$$
$$(T+1)\cdot\tau_{max}\cdot(1+\rho)^2 - \tau_{max} + \Delta,$$

all other correct processors have also concluded the first communication phase of their sub-ICA. This concludes Lemma 3.15.1. ❑

LEMMA 3.15.2.
After communication phase i $(1 \le i \le T)$, in real-time, the sub-ICAs of all correct processors are synchronized within a real-time bound of $(L_{i+1} / (1+\rho)) - \tau_{max}$.

Proof:
The proof is similar to that of Lemma 3.12.2. However, notice that, in Lemma 3.12.2., the lengths of the communication phases of the ICA as measured by the processor clock of any correct processor are given by A3.17 instead of by A3.15. (as in Lemma 3.12.2.)

We will prove this lemma by induction on *i*.

Basis: *i* = 1.
From Lemma 3.15.1. we know that after communication phase 1, the sub-ICAs of all correct processors have been synchronized within a real-time bound of $(T+1)\cdot\tau_{max}\cdot(1+\rho)^2 - \tau_{max} + \Delta$, which is equal to $(L_2 / (1+\rho)) - \tau_{max}$.

Induction step: *i* > 1.
We assume that Lemma 3.15.2. holds for $j = i - 1$. Thus, the induction hypothesis is:
After communication phase i - 1(2 ≤ i ≤ T), in real-time, the sub-ICAs of all correct processors are synchronized within a real-time bound of $L_i / (1+\rho) - \tau_{max}$.

Now, we prove that Lemma 3.15.2. also holds for $j = i$.

Notice that the maximal real-time difference in synchronization occurs between the sub-ICA of a correct processor *s* which runs a factor (1+ρ) slower than real-time and that of a correct processor *f* which runs a factor (1+ρ) faster than real-time. The real-time difference is maximal if processor *f* immediately detects the end of its communication phase, whereas processor *s* only detects it after real-time Δ.

For all *i* (with $1 \le i \le T+1$) and any correct processor *p*, let $L_i(p)$ be the real-time length of communication phase *i* in *p*. For a correct processor *s* that runs a factor (1+ρ) slower than real-time, the maximum real-time length of communication phase *i* in *s* is $L_i(s) = L_i \cdot (1+\rho) + \Delta$. For a correct processor *f* that runs a factor (1+ρ) faster than real-time, the minimum real-time length of communication phase *i* in *f* is $L_i(f) = L_i / (1+\rho)$. So, the real-time difference in protocol synchronization increases by an amount less than or equal to $L_i(s) - L_i(f)$. From the induction hypothesis, we know that the real-time difference after communication phase *i* - 1 is less than or equal to $L_i / (1+\rho) - \tau_{max}$, which is equal to $L_i(f) - \tau_{max}$. After communication phase *i*, the maximal real-time difference in protocol synchronization is $(L_i(f)-\tau_{max})+(L_i(s)-L_i(f)) = L_i(s) - \tau_{max}$, which is equal to $L_i \cdot (1+\rho) + \Delta - \tau_{max}$. Since, by A3.17, for $i > 1$, $L_{i+1} = [L_i \cdot (1+\rho) + \Delta]\cdot (1+\rho)$, the maximal real-time difference in protocol synchronization after communication phase *i* is $L_i \cdot (1+\rho) + \Delta - \tau_{max} = L_{i+1} / (1+\rho) - \tau_{max}$. This proves Lemma 3.15.2. ❑

LEMMA 3.15.3.
Let $m(mv,(\underline{\mathbf{p}}))$ be any valid message which is timely received and accepted by a correct processor *c* in communication phase *i* ($1 \le i \le T$) of *c*'s sub-ICA and assume that *c* relays $m(mv,(\underline{\mathbf{p}}))$ to a correct processor *d*, i.e., processor *c* sends $m(mv,(\underline{\mathbf{p}};d))$ to *d*. Then, message $m(mv,(\underline{\mathbf{p}};d))$ arrives in processor *d* in or before communication phase *i*+1 of *d*'s sub-ICA.

Proof:
Identical to the proof of Lemma 3.6.3. for authenticated self-synchronizing BAPs, except for the fact that processor c and d execute a sub-ICA instead of a sub-BAP, and the lengths of the communication phases of the sub-ICA executed on a correct processor, as measured by the processor clock of that correct processor, are given by A3.17 (whereas, in Lemma 3.6.3., the lengths of the various communication phases of the sub-BAP executed on a correct processor, as measured by the processor clock of that correct processor are given by A3.9). ❏

THEOREM 3.15.
If a correct processor accepts a valid message of a BAP belonging to the ICA with a certain message value *mv*, then all correct processors will accept a valid message of this BAP with message value *mv* before the end of the ICA.

Proof:
Identical to the proof of Theorem 3.9. for optimized authenticated self-synchronizing BAPs, except for the fact that the correct processors execute an ICA instead of a BAP. ❏

With the help of Theorem 3.14. and 3.15., it is possible to prove that the optimized authenticated self-synchronizing ICA presented in Section 3.4.2. satisfies conditions ICA1 and ICA2, which have been stated in Section 3.1.2. as follows:

ICA1. All correct processors agree on the initial value they think they have received from each of the processors.
ICA2. For every processor *p*, it holds that if *p* is correct, the above-mentioned agreement equals the initial value actually sent by processor *p*.

The proof is given by the following theorem.

THEOREM 3.16.
Provided that the unforgeability properties UP1 and UP2 from Section 3.3.3. and assumptions A3.1. through A3.3., A3.5 and A3.6., and A3.10 through A3.13. from Section 3.3.8. and A3.16 from Section 3.4.1. and A3.17 in this section hold, the optimized authenticated self-synchronizing ICA presented in this section satisfies conditions ICA1 and ICA2.

Proof:
We first prove that the optimized authenticated self-synchronizing ICA satisfies ICA1.

As stated before, an optimized authenticated self-synchronizing ICA runs on a system, *N*, of *N* processors, and consists of execution of *N* optimized authenticated self-synchronizing BAPs, one per processor.

From Theorem 3.14., we know that if a correct processor accepts a valid message of a BAP belonging to the ICA with a certain message value *mv*, then all correct processors will accept a valid message of this BAP with message value *mv* before the end of this ICA. Hence, for every BAP belonging to the ICA, the sets of message values of valid messages of the BAP accepted during the ICA are equal for all correct processors.

The decision taken in a correct processor about the initial value sent by the source of a certain BAP belonging to the ICA is based on applying a decision function on the set of message values of the valid messages of the BAP accepted during the multicast process of the ICA. Since all correct processors apply the same decision function on the same set of message values, the decisions of all correct processors are the same for every BAP belonging to the ICA, and thus, condition ICA1 is satisfied.

We now prove that the optimized authenticated self-synchronizing ICA satisfies ICA2.

We prove that in an arbitrary BAP belonging to the ICA, if the source *s* of the BAP holds an initial value *mv*, and the source is correct, then the decision about this BAP belonging to the ICA in any correct processor will be equal to *mv*.

We assume that the source *s* of a certain BAP belonging to the ICA is correct. This implies that the source initiates the BAP belonging to the ICA by sending a valid message $m(mv,(s;x))$ to any processor *x* in the set $S_{(s)}$, i.e. the set of processors to which the source should send a message, according to the subset selection rules of the BAP. From A3.12., we know that the subset selection rules are chosen such that any message from the source will be relayed to any correct processor in the system within at most *T* relay steps.

Furthermore, the source accepts a copy of the message itself in order to use it in the decision-making process of the BAP. From Theorem 3.14., we know that, if the source *s* which initiates the BAP is *correct*, any valid message sent from the source *s* to any correct processor *c* within at most *T* relay steps is timely received and accepted in processor *c*. Hence, before the end of the BAP, any correct processor *c* will accept a valid message $m(mv,(s;\underline{\mathbf{p}};c))$ received (directly or indirectly) from the source *s*.

For any correct processor *c*, the set of message values on which *c* will base its decision of the BAP belonging to the ICA will thus at least contain message value *mv* of the message received from the source. However, the set of message values on which *c* bases its decision of the BAP belonging to the ICA cannot contain other message values $mv' \neq mv$, because the source *s* is correct and hence, only generates and multicasts valid messages with message value *mv*, and, by the unforgeability properties UP1 and UP2, faulty processors are not able to generate and multicast valid messages $m(mv',(s;\underline{\mathbf{p}};c))$ with $mv' \neq mv$, since such messages need the signature of processor *s*, which the faulty processors are unable to produce, since *s* is correct. Hence, any correct processor (including the source itself, since it has accepted a copy of its own message) will decide *mv*, and thus, interactive consistency condition IC2 (the validity condition) is satisfied.

Since we have investigated the decision-making process in an arbitrary BAP belonging to the ICA, we may conclude that the above holds for any BAP belonging to the ICA. Hence, condition ICA2 is satisfied.

This concludes the proof of Theorem 3.16. ❏

3.4.3. A comparison of authenticated self-synchronizing ICAs and optimized authenticated self-synchronizing ICAs

In this section, for some practical values of T, N, ρ, τ_{min}, and τ_{max}, we compare the optimized authenticated self-synchronizing ICAs (described in Section 3.4.2.) with the authenticated self-synchronizing ICAs (described in Section 3.4.1.), which we will refer to as non-optimized self-synchronizing ICAs[50].

We choose $\rho=10^{-4}$ and $\tau_{max} = 20$ ms. These values are realistic for an ATM-network. Furthermore, we choose $\Delta = 20$ ms.

In Table 3.10 and 3.11, for some values of T and N, the required number of messages for both non-optimized and optimized authenticated self-synchronizing ICAs are given. We assume that, in any BAP in our optimized authenticated self-synchronizing ICAs, in every communication phase except the last one, a valid message is relayed to a subset of T+1 processors.

Notice that, since any ICA consists of execution of N BAPs, the required number of messages in an ICA is simply N times the number of messages required in a BAP (cf. Section 3.3.11.). Thus, for a non-optimized authenticated self-synchronizing ICA, the required number of messages *mess* is given by [51]

$$mess = N \cdot \sum_{i=1}^{T+1} i! \cdot \begin{bmatrix} N-1 \\ i \end{bmatrix}$$

whereas for an optimized authenticated self-synchronizing ICA, for $N \geq 2T+1$, the required number of messages *mess* is given by [52]

$$mess = N \cdot \left(\left(\sum_{i=1}^{T} (T+1)^{i} \right) + (T+1)^{T} \cdot (N-T-1) \right)$$

and for $T+2 \leq N \leq 2T$, by

$$N \cdot \left(\sum_{i=1}^{N-T-1} (T+1)^{i} \right) +$$
$$N \cdot \left(\sum_{i=N-T}^{T+1} (T+1)^{N-T-1} \cdot (i-N+T+1)! \cdot \begin{bmatrix} T \\ i-N+T+1 \end{bmatrix} \right)$$

The results show that for $N > T+2$, optimized authenticated self-synchronizing ICAs require fewer messages than non-optimized ones.

50. We do not compare the optimized self-synchronizing ICAs to other ICAs discussed in this chapter, since the latter ICAs are based on a system model with more restrictive synchronicity assumptions.
51. The BAPs that together make up the non-optimized ICA are a self-synchronizing version of the Lamport-protocol. See also Section 3.2.4.2.
52. The BAPs that together make up the optimized ICA are a self-synchronizing version of the Minimum Direction protocol presented in Section 3.2.3.1. and 3.2.4.4.

In Table 3.12, for some values of T, we have compared the maximum algorithm execution time (per processor) of both types of ICAs. We see that for $T \geq 2$, in an optimized ICA, the maximum execution time of a sub-ICA is greater than in a non-optimized ICA. The maximum algorithm execution time of either ICA is given by [53]

$$maxexecutiontime = (1+\rho) \cdot \sum_{k=1}^{T+1} L_k$$

where L_k is the length of communication phase k ($1 \leq k \leq T+1$) of the sub-ICA (of the corresponding type of ICA) of a certain processor p, as viewed from p.

Since $\tau_{max} \geq \tau_{min} \geq 0$ (by A3.1), and $\rho \geq 0$ (by A3.2), the execution time of an authenticated self-synchronizing ICA (respectively optimized authenticated self-synchronizing ICA) is always greater than or equal to that of an authenticated self-synchronizing BAP (respectively optimized authenticated self-synchronizing BAP)
.

Table 3.10: Required number of messages for T=1

N	#mess. in non-optimized ICAs	#mess. in optimized ICAs
3	12	12
4	36	24
5	80	40
16	4096	480

Table 3.11: Required number of messages for T=2

N	#mess. in non-optimized ICAs	#mess. in optimized ICAs
4	60	60
5	200	150
6	510	234
16	47280	2064

53. The algorithm execution time *executiontime* measured by a processor clock of any correct processor is equal to

$$executiontime = \sum_{k=1}^{T+1} L_k$$

However, since the processor clock of a correct processor may run up to a factor $(1+\rho)$ slower than real-time, the maximum algorithm execution time *maxexecutiontime* is equal to *executiontime*$\cdot(1+\rho)$.

Table 3.12: Algorithm execution time of sub-ICA (in ms)

T	Execution time in non-optimized ICAs	Execution time in optimized ICAs
1	100.028	100.028
2	180.064	240.096
3	280.120	440.200
4	400.200	700.380

3.5. References

[Alst96] Alstein, D., **Distributed Algorithms for hard real-time systems**, Ph.D. thesis, Eindhoven University of Technology, 1996.

[BaDo88] Bar-Noy, A., and Dolev, D., Families of Consensus Algorithms, in: Reif, J.H. (ed.), **VLSI Algorithms and Architectures, Proceedings of the 3rd Aegean Workshop on Computing, AWOC 88**, Corfu, Greece, 1988, pp. 380-390.

[BaMD93] Barborak, M., Malek, M., and Dahbura, A., The Consensus Problem in Fault-Tolerant Computing, in: **ACM Computing Surveys**, Vol.25, No.2, June 1993, pp. 171-220.

[BDGK95] Bar-Noy, Deng, X., Garay, J.A., and Kameda, T., Optimized Amortized Distributed Consensus, in: **Information and Computation**, Vol 120, No.1, 1995, pp. 93-100.

[BeGP89] Berman, P., Garay, J.A., and Perry, K.J., Towards Optimal Distributed Consensus (Extended Abstract), in: **Proceedings of the 30th Annual Symposium on Foundations of Computer Science**, IEEE Computer Society Press, 1989, pp. 410-415.

[BeGP92] Berman, P., Garay, J.A., and Perry, K.J., Optimal Early Stopping in Distributed Consensus (Extended Abstract), in: **Proceedings of the 6th International Workshop on Distributed Algorithms**, LNCS 647, Springer-Verlag, 1992, pp.221-237.

[BenO83] Ben-Or, M., Another advantage of free choice: Completely asynchronous agreement protocols, in: **The 2nd Annual ACM Symposium on Principles of Distributed Computing**, ACM, New York, 1983, pp.27-30.

[BrTo83] Bracha, G., and Toueg, S., Resilient consensus protocols, in: **The 2nd Annual ACM Symposium on Principles of Distributed Computing**, ACM, New York, 1983, pp.12-26.

[CASD95] Cristian, F., Aghili, H., Strong, R., and Dolev, D., Atomic broadcast: From simple message diffusion to Byzantine Agreement, in: **Information and Computation**, Vol.118, 1995, pp.158-179. (An earlier version of this paper appeared in: **Proceedings of the 15th International Symposium on Fault-Tolerant Computing**, Ann Arbor, June 1985, pp.200-206)

[CDDS85] Coan, B.A., Dolev, D., Dwork, C., and Stockmeyer, L., The Distributed Firing Squad Problem, in: **The 17th ACM Symposium on the Theory of Computing**, ACM, New York, 1985, pp.335-345.

[Coan90] Coan, B.A., Bibliography for Fault-Tolerant Distributed Computing, in: Simons, B., and Spector, A. (Eds.), **Fault-Tolerant Distributed Computing**, Springer Verlag, Berlin, 1990, ISBN 3-540-97385-0, pp.274-298.

[CoWe92] Coan, B.A., and Welch, J.L., Modular Construction of a Byzantine Agreement protocol with optimal message bit complexity, in: **Information and Computation**, Vol. 97, No.1, 1992, pp.61-85.

[Cris89] Cristian, F., Probabilistic clock synchronization, in: **Distributed Computing**, Vol. 3, 1989, pp.146-158.

[DHSS95] Dolev, D., Halpern, J.Y., Simons, B., and Strong, R., Dynamic Fault-Tolerant Clock Synchronization, in: **Journal of the ACM**, Vol.42, No.1, January 1995, pp.143-185. (An earlier version of this paper from the same authors was entitled: Fault-Tolerant Clock Synchronization, and appeared in: **Proceedings of the 3rd Annual ACM Symposium on Principles of Distributed Computing**, ACM, New York, 1984, pp.89-102).

[DiHe76] Diffie, W., and Hellman, M.E., New directions in cryptography, in: **IEEE Transactions on Information Theory**, IT22(6), November 1976, pp.644-650.

[DoDS87] Dolev, D., Dwork, C., and Stockmeyer, L., On the Minimal Synchronism Needed for Distributed Consensus, in: **Journal of the ACM**, Vol.34, No.1, January 1987, pp.77-97.

[DoHS86] Dolev, D., Halpern, J.Y., and Strong, H.R., On the possibility and Impossibility of Achieving Clock Synchronization, in: **Journal of Computer and System Sciences**, Vol.32, No.2, pp.230-250, 1986. (A preliminary version of this paper appeared in: **Proceedings of the 16th Annual ACM Symposium of Theory of Computation**, Washington D.C., 1984).

[DoRe85] Dolev, D., and Reischuk, R., Bounds on Information Exchange for Byzantine Agreement, in: **Journal of the ACM**, Vol.32, No.1, January 1985, pp.191-204.

[DoSt83] Dolev, D., and Strong, H.R., Authenticated algorithms for Byzantine Agreement, in: **Siam Journal on Computing**, Vol.12, No.4, 1983, pp. 656-666.

[DwLS88] Dwork, C., Lynch, N., and Stockmeyer, L., Consensus in the presence of partial synchrony, in: **Journal of the ACM**, Vol.35, No.2, April 1988, pp.288-323.

[FiLP85] Fischer, M., Lynch, N., and Paterson, M., Impossibility of distributed consensus with one faulty process, in: **Journal of the ACM**, Vol.32, No.2, April 1985, pp.374-382.

[FiLy82] Fischer, M., and Lynch, N., A lower bound for the time to assure interactive consistency, in: **Information Processing Letters**, Vol.14, No.4, June 1982, pp.183-186.

[GaMo93] Garay, J.A., and Moses, Y., Fully polynomial Byzantine Agreement in t+1 rounds, in: **Proceedings of the 25th Symposium on Theory of Computing**, 1993, pp. 31-41.

[GoLR95] Gong, L., Lincoln, P., and Rushby, J., Byzantine Agreement with Authentication: Observations and Applications in Tolerating Hybrid and Link Faults, in: **Proceedings of the 5th International Working Conference on Dependable Computing for Critical Applications DCCA-5**, Urbana-Champaign, Illinois, 1995, preprints, pp.79-90.

[HaMM85] Halpern, J., Megiddo, N., and Munshi, A., Optimal Precision in the Presence of Uncertainty, in: **Journal of Complexity**, Vol.1, 1985, pp.170-196.

[Jalo94] Jalote, P., **Fault Tolerance in Distributed Systems**, Prentice Hall, New Jersey, 1994, pp. 97-99.

[Krol91] Krol, Th., **A Generalization of Fault-Tolerance Based on Masking**, PhD. thesis, Eindhoven University of Technology, 1991.

[Krol95] Krol, Th., Interactive Consistency Algorithms Based on Voting and Error-Correcting Codes, in: **The 25th International Symposium on Fault-Tolerant Computing, Digest of Papers, FTCS-25 Silver Jubilee**, IEEE Computer Society Press, Los Alamitos, California, June 1995, pp.89-98.

[LaSP82] Lamport, L., Shostak, R. and Pease, M., The Byzantine Generals Problem, in: **ACM Transactions on Programming Languages and Systems**, Vol.4, No.3, July 1982, pp. 382-401.

[LuLy84] Lundelius, J., and Lynch, N., An Upper and Lower Bound for Clock Synchronization, in: **Information and Control**, Vol. 62, No. 2 /3, Aug. / Sept. 1984, pp. 190-204.

[LuLy88] Lundelius Welch, J., and Lynch, N., A New Fault-Tolerant Algorithm for Clock Synchronization, in: **Information and Computation**, Vol.77, 1988, pp. 1-36.

[Merk82] Merkle, R.C., **Secrecy, Authentication, and Public Key Systems**, UMI Research Press, Michigan, 1982. A revision of 1979 Ph.D. thesis at Stanford University.

[Merk89] Merkle, R.C., One Way Hash Functions and DES, in: **Advances in Cryptology - Proceedings of CRYPTO’89**, LNCS 435, Springer-Verlag, 1989, pp.428-446.

[Mull93] Mullender, S.J., **Distributed systems**, 2nd Edition, Addison-Wesley, New York, 1993, p.533.

[Nieu91] Nieuwenhuis, L.J.M., **Fault Tolerance through Program Transformation**, Ph.D. thesis, University of Twente, Enschede, 1991.

[PeSL80] Pease, M., Shostak, R., and Lamport, L., Reaching agreement in the presence of faults, in: **Journal of the ACM**, Vol.27, No.2, April 1980, pp.228-234.

[PoKr95] Postma, A., and Krol, Th., Interactive Consistency Algorithms Based on Authentication and Error-Correcting Codes, in: **Proceedings of the 5th International Working Conference on Dependable Computing for Critical Applications DCCA-5**, Urbana-Champaign, Illinois, 1995, preprints, pp.91-101.

[PoKr96] Postma, A., and Krol, Th., Interactive Consistency in Quasi-Asynchronous Systems, in: **Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems ICECCS’96**, Montréal, Canada, October 21-25, 1996, pp.2-9. (A preliminary version of this paper appeared as: Memoranda Informatica 96-05, University of Twente, Enschede, March 1996.)

[PoKM97] Postma, A., Krol, Th., and Molenkamp, E., Optimized Authenticated Self-synchronizing Byzantine Agreement Protocols, in: **Proceedings of the 1997 Pacific Rim International**

Symposium on Fault Tolerant Systems (PRFTS), December 15-16, 1997, Taipei, Taiwan, IEEE Computer Society Press, Los Alamitos, California, 1997, ISBN 0-8186-8212-4, pp. 122-129.

[Ponz91] Ponzio, S., Consensus in the Presence of Timing Uncertainty: Omission and Byzantine Failures (extended abstract), in: **Proceedings of the 10th Annual ACM Symposium on Principles of Distributed Computing**, Montréal, Canada, 1991, pp.125-138.

[Post96] Postma, A., **Implementation of a simulation program of a self-synchronizing Byzantine Agreement Protocol**, Memoranda Informatica 96-19, University of Twente, Enschede, November 1996.

[Rabi83] Rabin. M.O., Randomized Byzantine Generals, in: **Proceedings of the 24th Symposium on Foundations of Computer Science**, 1983, pp.403-409.

[Rabi89] Rabin, M.O., Efficient Dispersal of Information for Security, Load Balancing, and Fault Tolerance, in: **Journal of the ACM**, Vol.36, No.2, April 1989, pp.335-348.

[RiSA78] Rivest, R., Shamir, A., and Adleman, L., A method for obtaining digital signatures and public-key cryptosystems, in: **Communications of the ACM**, Vol. 21, 1978, pp.120-126.

[Schn90] Schneider, F.B., Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial, in: **ACM Computing Surveys**, Vol.22, No.4, December 1990, pp.299-319.

[SiLL90] Simons, B., Lundelius Welch, J., and Lynch, N., An Overview of Clock Synchronization, in: Simons, B., and Spector, A. (Eds.), **Fault-Tolerant Distributed Computing**, Springer Verlag, Berlin, 1990, ISBN 3-540-97385-0, pp.84-96.

[SuHW94] Suri, N., Hugue, M.M., and Walter, C.J., Synchronization Issues in Real-Time Systems, in: **Proceedings of the IEEE**, Vol.82, No.1, January 1994, pp.41-54.

[TuSh92] Turek, J., and Shasha, D., The Many Faces of Consensus in Distributed Systems, in: **IEEE Computer**, Vol. 18, No.6, June 1992, pp.8-17.

[VaOo89] Vanstone, S.A., and van Oorschot, P.C., **An Introduction to Error Correcting Codes with Applications**, Kluwer Academic Publishers, Boston, 1989.

CHAPTER 4

Distributed Cryptographic Function Application Protocols

"For jointly authored works, all the joint authors should sign, or one of the authors should sign as an authorized agent for the others."

IEEE Copyright Form

Abstract

A dependable data storage system should be fault-tolerant and secure in order to minimize the probability that data stored in such a system gets permanently lost or undetectably corrupted, or is read by unauthorized third parties. In such a system, all data is stored in a fault-tolerant and secure way on a set, N , of N processors, up to T of which are allowed to behave maliciously. A failure of a processor in N may lead to loss or corruption of data stored on it. It is desirable to recover this data as soon as possible. Whether lost or corrupted data can be recovered or not, should not depend on the correctness of a single processor. Therefore, a group of processors should be responsible for recovery of lost or corrupted data. Since data is stored in a secure way, the recovery process requires encryption of data with secret cryptographic functions. For this purpose, so-called distributed cryptographic function application protocols (DCFAPs), introduced in this chapter, can be used.

Distributed cryptographic function application protocols are executed on a set, N , of N processors, up to T of which may behave maliciously, in order to apply a secret cryptographic function F on a commonly known piece of data, B, while satisfying the following properties:

- ❏ *malicious behaviour of up to T processors from N does not inhibit application of cryptographic function F*
- ❏ *any group of N-T or more processors from N is capable of applying function F*
- ❏ *T or fewer colluding processors from N are unable to compute the secret cryptographic function F, or to apply F without the help of one or more correct processors.*

Notice that $N \geq 2T+1$ is a necessary requirement for the last two properties to be non-conflicting.

Application of a DCFAP guarantees that all correct processors in N reach agreement on the result of application of function F on the commonly known piece of data, B, provided that the number of maliciously behaving processors is less than or equal to T.

4.1. Introduction

Nowadays, computer systems are used for a wide range of applications. For some applications, it is of crucial importance to minimize the probability that data stored by users of the system gets permanently lost or corrupted. Two main causes for permanent loss or corruption of data exist [1]:

❑ failure of the processor(s) on which (part of) the data was stored

❑ undetected mutilation of (part of) the data by unauthorized third parties

In order to minimize the probability of permanent loss or corruption of data due to one of the above-mentioned causes, data should be stored in a ***fault-tolerant and secure*** way.

Most research in the area of ***fault-tolerance*** focuses on the design of ***fault-tolerant systems***, i.e., systems that maintain their system function (e.g., fault-tolerant data storage or fault-tolerant data communication) despite failures of processors in the system, whereas research in the field of ***cryptography*** is mainly concerned with the design of ***secure systems***, i.e., systems that maintain their system function (e.g., secure data storage or secure data communication) despite the actions of malicious attackers (i.e., unauthorized third parties).

In this chapter, for a data storage system, we combine both research areas in an approach to describe a data storage system that is both fault-tolerant and secure. For this purpose, so-called ***distributed cryptographic function application protocols*** (***DCFAPs***) can be used. The focus of this chapter will therefore be on DCFAPs.

The rest of this section is structured as follows. First, in Section 4.1.1., we describe how fault-tolerant data storage can be established. Then, in Section 4.1.2., we investigate how to establish secure data storage. In Section 4.1.3., both concepts are combined in order to achieve fault-tolerant and secure data storage. In Section 4.1.4., some related work is discussed. Finally, in Section 4.1.5., the structure of the rest of this chapter is indicated.

4.1.1. Fault-tolerant data storage

In order to make a data storage system resilient to failures of one or more of its processors, redundancy in the form of extra processors can be employed. Provided that all data in the system is stored in a fault-tolerant way as a collection of constituent data fragments of equal length on the disks of N processors, simultaneously occurring arbitrary (or Byzantine) failures of a limited number of processors need not lead to permanent loss of data. For this purpose, the collection of data fragments should contain sufficient ***redundancy***.

A simple method to obtain the required redundancy is by making a number of copies of the original data (i.e., every data fragment is just a copy of the original data). This method is called ***data replication***, and is widely used in commercial and experimental fault-tolerant systems. A more sophisticated method, which requires less redundancy

1. There is also a possibility that data gets accidentally lost or corrupted due to mistakes of the owner of the data. This possibility will not be further investigated in this chapter, however.

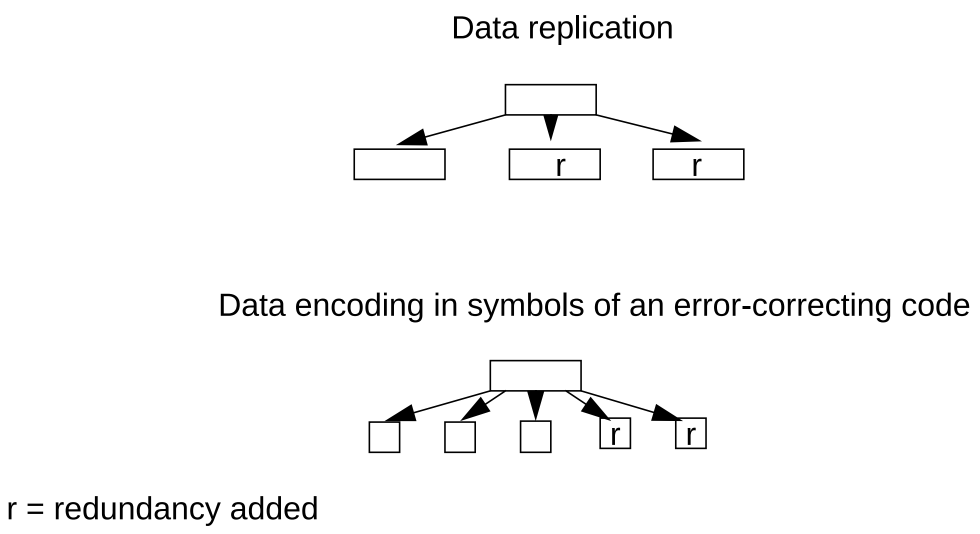


Figure 4.1. Data replication versus data encoding

than the method of data replication, is to ***encode the data into symbols of an error-correcting code***. See also Figure 4.1. In this method, every data fragment contains a number of symbols (e.g., bits or bytes) of an error-correcting code. Assume that every processor p_i ($1 \leq i \leq N$) possesses one data fragment D_i. Assume furthermore that data fragment D_i ($1 \leq i \leq N$) contains b symbols, numbered $s_{i,1}, \ldots s_{i,b}$. Then, for any j, with $1 \leq j \leq b$, the j-th code word of the error-correcting code consists of the j-th symbol of every data fragment, i.e., $s_{1,j} \ldots s_{N,j}$.

In fact, the method of data replication is a special case of data encoding, in which the data is encoded into symbols of a so-called ***repetition code***, i.e., an error-correcting code in which any symbol of the code word is just a copy of the original data. The method of encoding will be illustrated by means of the following example.

Example:
In this example we consider the encoding of a file into symbols of a 1-erasure-correcting code (i.e., a special kind of error-correcting code which is capable of correcting one erasure). It is important to notice that in this example we only consider one of the many ways in which a file may be encoded.

Assume a file with contents “THIS IS AN EXAMPLE” is stored as a collection of 6 data fragments D_i ($1 \leq i \leq 6$) on 6 processors. As a consequence, every code word of the applied 1-erasure-correcting code consists of 6 symbols. Each code word of the 1-erasure-correcting code should contain at least one redundant symbol in order to be able to correct one erasure [VaOo89]. For simplicity, we assume that all redundant symbols r_j ($1 \leq j \leq b$) are stored in one data fragment (D_6). The contents of the file are stored in the other data fragments. We assume that the file is padded with spaces until the number of symbols in the file is a multiple of 5 (i.e., a multiple of the number of data fragments onto which the file symbols are stored). In this way, each data fragment can be given an equal number of symbols, b.

The symbols of the file are stored in the data fragments as illustrated in Figure 4.2.

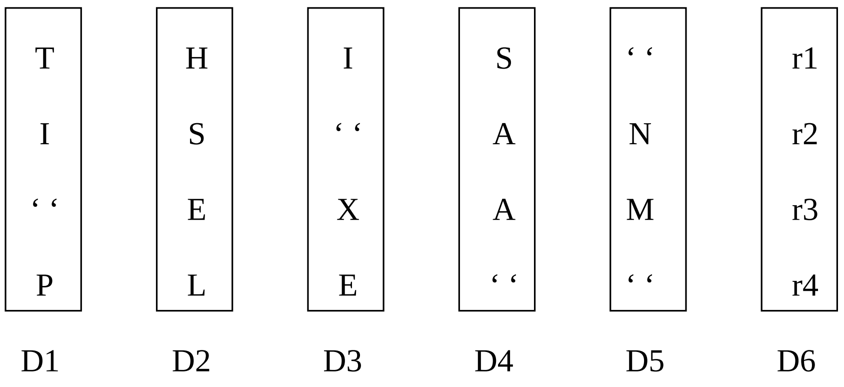


Figure 4.2. Encoding the data into data fragments

Assume that an 8-bit ASCII-code is used to encode each symbol. We assume that the modulo-2^8 sum of the symbols of any code word, i.e., the modulo-2^8 sum of any redundant symbol r_j ($1 \le j \le b$) with the symbols $s_{i,j}$ ($1 \le i \le 5$) yields the symbol 255 (i.e., we assume odd parity). The redundant symbols r_j can now easily be calculated. Namely, the modulo-2^8 sum of the symbols $s_{i,j}$ ($1 \le i \le 5$) and the symbol 255 yields symbol r_j. Figure 4.3. illustrates the calculation of r_1.

```
   T =          84
   H =          72
   I =          73
   S =          83
 ‘ ‘=           32
               255
___________________    + (mod 256)
  r1 =          87
```

Figure 4.3. Calculation of redundant symbol r_1

In an analogous way, a symbol $s_{i,k}$ (for some k, with $1 \le k \le b$) that is lost due to loss or corruption of one of the data fragments D_i (for some i, with $1 \le i \le 6$) can be recovered with the help of the other symbols $s_{j,k}$ (for all j, with $1 \le j \le 6$, and $j \ne i$) of the code word in the remaining data fragments. Namely, the modulo-2^8 sum of the symbols $s_{j,k}$ and the symbol 255 yields the lost symbol $s_{i,k}$. ❏

Provided that the collection of data fragments contains sufficient redundancy, any data fragments that get lost or corrupted due to one or more processor failures may be recovered with the help of the remaining data fragments. After replacement of the failed processor and recovery of all lost or corrupted data fragments that were stored on its disk, the system is able to survive a subsequent processor failure. Thus, it is advantageous to recover lost data fragments (and replace failed processors) as soon as possible.

4.1.2. Secure data storage

In a multi-user environment, users may want to protect their confidential data against unauthorized reading and undetected mutilation by others. In order to solve these security problems, cryptography should be applied.

Cryptography (see, e.g., [Simm92]) is the study of "mathematical" systems for solving two kinds of security problems: ***privacy*** and ***authentication*** [DiHe76]. Solutions to the *privacy problem* aim to inhibit extraction of information by unauthorized parties from a certain piece of confidential data communicated or stored in a system. Solutions to the *authentication problem* try to guarantee a possibility to verify the identity of the creator of a certain piece of data communicated or stored in a distributed system, provided that the data is uncorrupted. Furthermore, authentication provides a means for ***mutilation detection***, i.e., authentication offers a possibility to detect mutilations of the data by unauthorized parties.

Usually, cryptographic techniques are applied to protect data while it is *communicated* between two or more users (also called ***principals***) in a distributed system. These techniques can, however, be equally well applied to protect data *stored* on a computer against malicious actions. In [Desm94], the aforementioned applications of cryptography are defined as ***communication security*** and ***computer security***, respectively.

Protection of data stored in a data storage system against unauthorized reading and undetected mutilation requires a solution to both the privacy problem and the authentication problem.

The rest of this section is structured as follows. In Section 4.1.2.1., an introduction to cryptography is given. Readers familiar with the basic principles of cryptography may skip this section. In Section 4.1.2.2., we consider how the data can be protected from being read by unauthorized third parties. Then, in Section 4.1.2.3., it is investigated how data can be protected from undetectably being mutilated by unauthorized third parties. Finally, in Section 4.1.2.4., we investigate how the former two techniques can be combined in order to protect data against both unauthorized reading and undetected mutilation.

4.1.2.1. An introduction to cryptography

Cryptography has a long and colourful history and dates back to the days of Julius Caesar. Its original goal was to guarantee a possibility for two parties to exchange confidential information by means of sending messages to each other, without giving third parties the opportunity to retrieve any sensible information from possibly intercepted messages. Thus, the two parties had to take care that any confidential information was 'hidden' in the messages exchanged. For this purpose, the two parties transformed any piece of confidential information they wanted to communicate into some unreadable form. Any message communicated between these two parties contained only information in unreadable form. For the receiving party to be able to extract the confidential information again from the received message, a priori, the two communicating parties had agreed upon a way to transform any incoming message into a readable piece of information.

In general, in order to protect a piece of data from being read by unauthorized third parties, the original data (also called the ***plaintext***) is transformed into some unreadable form (called ***ciphertext***) by means of a so-called enciphering transformation (or ***encryption function***). The inverse transformation also exists and is called the deciphering transformation (or ***decryption function***). This transformation is only known to authorized parties. By executing the deciphering transformation on a piece of ciphertext, the corresponding plaintext can be obtained.

The process of execution of an enciphering transformation on a piece of plaintext P is called the ***encryption*** of P. Conversely, ***decryption*** of a ciphertext C means execution of a deciphering transformation on C. Encryption and decryption of data is usually done by executing a commonly known algorithm (called the ***cryptographic algorithm***). The output of the cryptographic algorithm depends not only on the input data, but also on a so-called ***cryptographic key*** fed into the algorithm. Such a cryptographic key consists of a (usually) short string of characters that selects one of many potential enciphering (respectively deciphering) transformations. The cryptographic key is selected from a finite set called the ***keyspace***. Usually, for encryption and decryption of data, the same cryptographic algorithm is applied, only the supplied cryptographic key is different. A ***cryptographic function*** is specified as a cryptographic algorithm fed with a certain cryptographic key.

In [DiHe76], the notion of a ***cryptographic system*** (or ***cryptosystem***) was introduced. In accordance with [DiHe76], a cryptographic system can be defined as follows:

D4.1. Let *Keyspace* be the keyspace from which cryptographic keys are selected, and let *Plaintext*, respectively *Ciphertext* be the set of all plaintext messages respectively ciphertext messages. Then, a ***cryptographic system*** is a set

$$\{S_k \mid k \in \mathit{Keyspace}\}$$

of invertible transformations

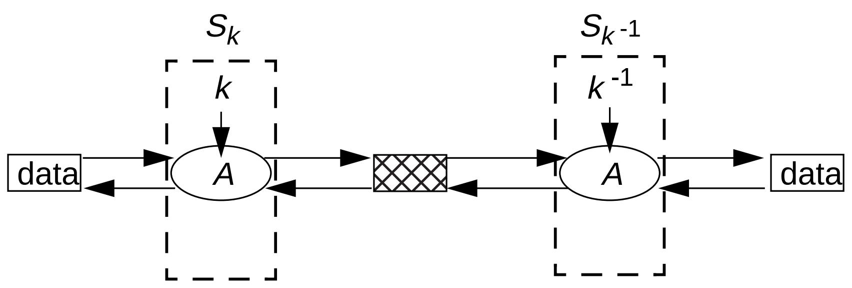
$$S_k\colon \mathit{Plaintext} \rightarrow \mathit{Ciphertext}$$

Here, the enciphering transformation is denoted as S_k (for some key $k \in \mathit{Keyspace}$). The corresponding deciphering transformation is denoted by $S_{k^{-1}}$ (for the corresponding key $k^{-1} \in \mathit{Keyspace}$) [2]. For any $k, k^{-1} \in \mathit{Keyspace}$, the transformations S_k and $S_{k^{-1}}$ are each other's inverses, i.e., for any $P \in \mathit{Plaintext}$ and any $C \in \mathit{Ciphertext}$ it holds that:

$$C = S_k(P) \Leftrightarrow P = S_{k^{-1}}(C)$$

The idea of a cryptosystem is illustrated in Figure 4.4.

2. In the sequel, for pairs of cryptographic functions S_k and $S_{k^{-1}}$ we will also use the shorthand notation S and $S^{(-1)}$.



$S_k, S_{k^{-1}}$ = cryptographic functions
k, k^{-1} = cryptographic keys
A = cryptographic algorithm

Figure 4.4. Idea of a cryptosystem

Roughly, two types of cryptosystems exist: symmetric (or secret-key) cryptosystems and asymmetric (or public-key) cryptosystems.

A ***symmetric cryptosystem*** (or ***secret-key cryptosystem***) is a cryptosystem in which the same cryptographic key is used for both the enciphering and the deciphering transformation. Conversely, in an ***asymmetric cryptosystem*** (or ***public-key cryptosystem***), for encryption and decryption respectively, different cryptographic keys are used. Both types of cryptosystems will be briefly discussed below.

Symmetric cryptosystems
In a *symmetric cryptosystem* the same cryptographic key is applied for encryption and decryption, and this key is kept secret. Without knowledge of the secret cryptographic key used in a symmetric cryptosystem, it should be computationally infeasible to obtain the plaintext from ciphertext encrypted with this cryptographic key. For practical implementations of cryptosystems, Shannon, in [Shan49], suggested two general principles, ***diffusion*** and ***confusion***. Diffusion is defined as the spreading out of the influence of a single plaintext digit over many ciphertext digits so as to hide the statistical structure of the plaintext. Confusion is defined as the use of enciphering transformations that complicate the determination of how the statistics of the ciphertext depend on the statistics of the plaintext. Following these principles contributes to the design of cryptosystems that are hard to break.

A well-known example of a symmetric cryptosystem (designed in accordance with Shannon's diffusion and confusion principles) is the so-called ***Data Encryption Standard*** (or DES for short), published in [FIPS77]. Descriptions of the DES-algorithm can be found in many textbooks on cryptography (e.g., [Simm92]) A short good description is given in [DiHe79]. In DES, the cryptographic algorithm consists of several simple arithmetical and bit operations, like addition and subtraction, and shifting and selecting bits, each of which can be executed very fast. Fast hardware implementations of DES are commercially available. Indeed, an implementation of a 1Gbit/s DES chip has been described in [Eber93].

Asymmetric cryptosystems

The concept of an *asymmetric cryptosystem* received wide attention through the paper written by Diffie and Hellman [DiHe76]. Asymmetric cryptosystems separate the capacities for encryption and decryption so that, either many users can encrypt messages in such a way that only one user can read them, or one user can encrypt messages in such a way that many users can read them. In an asymmetric cryptosystem, for every user *u*, two cryptographic functions $K_{(u)}$ and $K_{(u)}^{(-1)}$ are defined 3, which are each other's inverses. Each user *u* makes the cryptographic key of the function $K_{(u)}^{(-1)}$ known to all other users, whereas it keeps the cryptographic key needed to perform function $K_{(u)}$ secret. An important requirement is that, without knowledge of the secret cryptographic key (the so-called ***trapdoor information***) needed to perform $K_{(u)}$, it should be computationally infeasible to obtain $K_{(u)}$ from $K_{(u)}^{(-1)}$, or from any data encrypted with $K_{(u)}^{(-1)}$. Then, in order to protect a piece of data from unauthorized reading, it is encrypted with the publicly known cryptographic function $K_{(u)}^{(-1)}$. Since the cryptographic key needed to perform function $K_{(u)}$ is kept secret, unauthorized third parties are incapable of reading data encrypted with function $K_{(u)}^{(-1)}$. On the other hand, it should be easy to obtain $K_{(u)}$, once the secret cryptographic key needed to perform $K_{(u)}$ is known. Any function $K_{(u)}$ satisfying the above requirements is called a ***trapdoor one-way function***.

Trapdoor one-way functions were originally introduced in [DiHe76], and based on the concept of ***one-way functions***, proposed by Needham in [Wilk72]. In [DiHe76], a *one-way function f* is defined as a function, of which, for any argument *x* in its domain, it is easy to compute the corresponding value $f(x)$, yet, for almost all *y* in the range of *f*, it is computationally infeasible to solve the equation $y = f(x)$ for any suitable argument *x*. Diffie and Hellman in [DiHe76] state that trapdoor one-way functions are not really one-way in that simply computed inverses exist. But given an algorithm for the forward function, it is computationally infeasible to find a simply computed inverse. Only through knowledge of certain trapdoor information, one can easily find the easily computed inverse. In [Simm92, p.25] a *trapdoor one-way function* is defined as a family of invertible functions f_z, indexed by *z*, such that, given *z*, it is easy to find algorithms E_z and D_z that easily compute $f_z(x)$ and $f_z^{(-1)}(y)$ for all *x* and *y* in the domain and range, respectively, of f_z; but for virtually all *y* in the range of f_z, it is computationally infeasible to compute $f_z^{(-1)}(y)$ even when one knows E_z.

Unfortunately, no one has yet produced a proof that any function is a one-way function or a trapdoor one-way function [Simm92, p.31]. Diffie and Hellman in [DiHe76] sug-

3. In the original paper on public-key cryptosystems [RiSA78], for any user *u*, his two cryptographic functions are denoted by D_u and E_u respectively. The function names D_u and E_u suggest that D_u is used for decryption and E_u for encryption. However, this is not always true, e.g., in any solution to the authentication problem (Section 4.1.2.2.), function D_u is used for signing (i.e., encryption of a hash value) and E_u for verification (i.e., decryption of a hash value). In order to avoid confusion, we will use the names $K_{(u)}$ and $K_{(u)}^{(-1)}$ instead. Function $K_{(u)}$ corresponds to D_u in [RiSA78], whereas function $K_{(u)}^{(-1)}$ corresponds to function E_u.

gested the use of NP-complete problems for practical implementations of trapdoor one-way functions. In literature, several implementations of public-key cryptosystems based on NP-complete problems have been proposed (e.g., [RiSA78, Merk78, McEl78, ElGa85], for an overview, see [Simm92, pp.135-285]). Some of these systems (e.g., the one proposed in [Merk78]) have been broken (i.e., attacks are possible by means of which the secret cryptographic function can be computed without knowledge of the secret cryptographic key). For details, see [Simm92, pp.501-558].

A well-known example of an implementation of an asymmetric cryptosystem is RSA (originally proposed in [RiSA78]). In RSA, function $K_{(u)}^{(-1)}$ consists of a complex arithmetical operation (discrete exponentiation), that requires a considerable computation overhead. As a result, RSA tends to be slow (i.e., requires long execution time).

4.1.2.2. Protection of data against unauthorized reading by others

As already stated in the previous section, in order to protect data from being read by unauthorized third parties, the original data should be transformed into some unreadable form by means of an encryption function. It is essential that the decryption function is only known to authorized parties.

For this purpose, for every user *u* in the system, an encryption and a decryption function are defined, which are each other's inverses. The encryption and decryption function of a user *u* aim to solve the *privacy* problem, and hence, they will be denoted by $P_{(u)}^{(-1)}$ and $P_{(u)}$, respectively.

If a *symmetric* cryptosystem is used for encryption and decryption, then $P_{(u)}^{(-1)} = P_{(u)}$, and both functions are kept secret. If an *asymmetric* cryptosystem is used for encryption and decryption, then the encryption function $P_{(u)}^{(-1)}$ is made public, whereas the decryption function $P_{(u)}$ is kept secret.

Protection of data against unauthorized reading can be done by encrypting the data with cryptographic function $P_{(u)}^{(-1)}$. Then, the data can only be read after it has been decrypted with $P_{(u)}$. Since $P_{(u)}$ is kept secret, and it is computationally infeasible to obtain $P_{(u)}$ from any data encrypted with $P_{(u)}^{(-1)}$, unauthorized parties (which do not know $P_{(u)}$) cannot read data which has been encrypted with $P_{(u)}^{(-1)}$. See Figure 4.5.

Encryption and decryption of data is done much faster with a symmetric cryptosystem than with an asymmetric cryptosystem. Therefore, usually, a symmetric cryptosystem is used to encrypt or decrypt bulk data in a system, whereas an asymmetric cryptosystem is usually applied to encrypt or decrypt small pieces of data, like, e.g., hash values (see next section).

4.1.2.3. Protection of data against undetected mutilation by others

In order to protect a certain piece of data against undetected mutilation by unauthorized parties, it should be impossible for a maliciously behaving processor to undetectably replace a piece of data, *B*, by a different piece of data, *B*'.

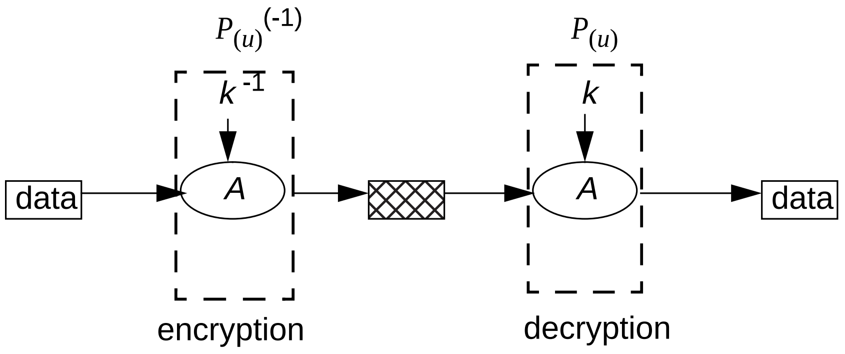


Figure 4.5. Protection of data against unauthorized reading

Since unauthorized processors may behave maliciously in order to mutilate data, simply adding some redundancy (like, e.g., a cyclic redundancy check) to the data is not sufficient to protect the data against undetected mutilation. In other words, processors should generate their pieces of data in such a way, that they can be distinguished from any piece of data generated by other processors. For this purpose, a processor should apply a secret cryptographic function.

Therefore, for any user u in the system, two different cryptographic functions are defined, which are each other's inverses. The cryptographic functions of any user u aim to solve the *authentication* problem, and hence, these functions will be denoted by $A_{(u)}$ and $A_{(u)}^{(-1)}$, respectively.

If a *symmetric* cryptosystem is used to protect data against undetected mutilation, then $A_{(u)}^{(-1)} = A_{(u)}$, and both functions are kept secret. If an *asymmetric* cryptosystem is used, then function $A_{(u)}^{(-1)}$ is made public, whereas $A_{(u)}$ is kept secret.

In order to be able to recover lost data fragments as soon as possible, it is desirable to be able to delegate this check to others[4]. Therefore, it is important that in order to perform the check, no knowledge about the semantics of the data is required. This implies that, for this check, besides the data, some redundant information should be supplied. Then, the check boils down to verifying if the data is compatible with the supplied redundant information. Inherently, it is impossible to distinguish between mutilations of the data and mutilations of the supplied redundant information. Since undetected mutilations of the data should be avoided, any mutilation that is detected is regarded as a mutilation of the data.

Any user u, which possesses a secret cryptographic function $A_{(u)}$ may protect a piece of data, B against undetected mutilation, in either of both ways:

4. I.e., the capability of performing the check should not depend on the correctness of the owner of the data (See also Section 4.1.3.4.)

1. By adding some redundancy, say $R(B)$, to B, and encrypting $B,R(B)$ with $A_{(u)}$, resulting in $A_{(u)}(B,R(B))$.
2. By adding $A_{(u)}(B)$ to B. This results in B, $A_{(u)}(B)$.

Notice that, in both alternatives, for unauthorized users which do not possess $A_{(u)}$, it is impossible to undetectably mutilate the data or the redundant information. However, it is impossible for user u to distinguish between mutilations of the data and mutilations of the corresponding redundant information.

In both alternatives, the entire data, B, has to be encrypted. Encrypting large amounts of data with $A_{(u)}$ is unattractive, because it endangers security (viz., malicious processors collecting sufficient amounts of data in encrypted form may obtain enough information in order to find $A_{(u)}$) and it may be time-consuming (since encryption usually requires a considerable computation overhead).

By encrypting only a representative part of the data, B, it is possible to reduce the amount of data that has to be encrypted.

A representative part of the data, B, may be obtained by applying a so-called ***hash function*** H on B, yielding a ***hash value*** (or ***message digest***) $H(B)$ of B. Such a hash value is usually a short fixed-length sequence of bits.

Ideally, the hash function should have the property, that for any data B with corresponding hash value $H(B)$, it is computationally infeasible to find data B', different from B, such that $H(B') = H(B)$, otherwise B can be undetectably changed into B'. As pointed out in [Merk89], the above requirement does not guarantee that it is also computationally infeasible to find pairs of input that map onto the same output. I.e., it might be easy for a malicious processor to find some inputs Z and Z', for which it holds that $H(Z) = H(Z')$. If X would (accidentally) be chosen to be equal to such a value Z or Z', it would be easy for a maliciously behaving processor to undetectably change X into X'. Thus, given H, it should be computationally infeasible to find any pair, X, X', such that $X \neq X'$, and $H(X) = H(X')$. Thus, hash function H should be a ***strong one-way hash function*** [5]. Strong one-way hash functions were first defined in [Merk82]. We adopt the definition of [Merk89], which states that a strong one-way hash function is a function H such that:

SOHF1. H can be applied to any argument of any size. For notational convenience, H applied to more than one argument is equivalent to H applied to the bit-

5. Merkle (in [Merk82]) was the first to define one-way hash functions and to distinguish between weak and strong one-way hash functions. In the definition of a weak one-way hash function, in contrast with the definition of a strong one-way hash function given above, the requirement that it should be impossible to find pairs of input that map onto the same output is omitted. In [Merk89], a ***weak one-way hash function*** is defined as a function H such that:
WOHF1. H can be applied to any argument of any size. For notational convenience, H applied to more than one argument is equivalent to H applied to the bit-wise concatenation of its arguments.
WOHF2. H produces a fixed-size output.
WOHF3. Given H and X, it is easy to compute $H(X)$.
WOHF4. Given H and a “suitably chosen” (e.g., random) X, it is computationally infeasible to find an $X' \neq X$ such that $H(X) = H(X')$.

wise concatenation of all its arguments.

SOHF2. *H* produces a fixed size output.

SOHF3. Given *H* and *X*, it is easy to compute *H*(*X*).

SOHF4. Given *H*, it is computationally infeasible to find any pair *X*, *X*', such that $X \neq X'$ and $H(X) = H(X')$.

Several strong one-way hash functions have been proposed, e.g., in [Davi83, Denn84, Merk89, Wint84]. For more details about one-way hash functions, the reader is referred to [Gong90].

Furthermore, it should be impossible to undetectably change *H*(*B*) (otherwise, a malicious processor may create *H*(*B*'), and then undetectably change *B* into *B*'). Therefore, either, the hash function *H* itself should be a secret cryptographic function, or *H*(*B*) should be encrypted with a secret cryptographic function. Since *H* may be a publicly known function, in this chapter, we assume that the hash value *H*(*B*) of a piece of data, *B*, is encrypted with the secret cryptographic function possessed by the owner of *B*.

Thus, any user *u*, with secret cryptographic function $A_{(u)}$ protects a piece of data, *B*, against undetected mutilation by storing it as

$$B, A_{(u)}(H(B)).$$

All processors that are capable of decrypting the encrypted hash value are capable to decide whether the data has been corrupted or not.

Protection of data against undetected mutilation is illustrated in Figure 4.6.

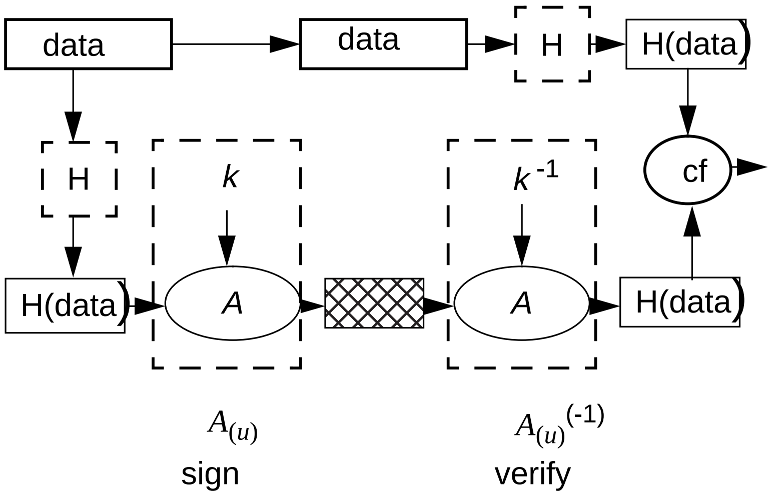


Figure 4.6. Protection of data against undetected mutilation

If a *symmetric cryptosystem* is applied for encryption of the hash value, then **only the owner** of the data is capable of checking whether the data has been corrupted or not

(Since the same cryptographic function is used for both encryption and decryption, the decryption function is also kept secret). If an *asymmetric cryptosystem* is applied for encryption of the hash value, then **any processor** in the system is capable of checking whether the data has been corrupted or not, since the publicly known cryptographic function $A_{(u)}^{(-1)}$ can be used to decrypt the hash value.

A processor is said to ***(digitally) sign*** a piece of data, B, if it adds an encrypted hash value of B to B. The process of checking whether or not a digitally signed piece of data, B, has been mutilated or not is called the ***verification*** of B.

4.1.2.4. Protection of data against unauthorized reading and undetected mutilation by others.

In order to protect data against both unauthorized reading and undetected mutilation by others, the techniques described in Section 4.1.2.2. and 4.1.2.3. should be combined.

Applying the same cryptographic function for encryption (decryption) and signing (verification) is generally considered as dangerous in security (although systems are often designed this way). Hence, since both techniques are combined in order to protect data against unauthorized reading and undetected mutilation by others, it is assumed that the cryptographic functions used for encryption (decryption) and signing (verification) are different. I.e., for any user u, $A_{(u)} \neq P_{(u)}$, and $A_{(u)}^{(-1)} \neq P_{(u)}^{(-1)}$.

Assume that a user u wants to protect a piece of data, B, against unauthorized reading and undetected mutilation by others. The techniques described in Section 4.1.2.2. and 4.1.2.3. can be combined in three ways:

Combination C1

Here, any piece of data is signed before the data is encrypted. The result can be denoted by

$$P_{(u)}^{(-1)}(B),\ A_{(u)}(H(B)) \qquad \square$$

Combination C2

Here, any piece of data is signed first. After that, the data, together with its signed hash value is encrypted. The result can be denoted by

$$P_{(u)}^{(-1)}(\ B,\ A_{(u)}(H(B))\ ) \qquad \square$$

Combination C3

Here, any piece of data is encrypted before it is signed. The result can be denoted by

$$P_{(u)}^{(-1)}(B),\ A_{(u)}(H(\ P_{(u)}^{(-1)}(B))) \qquad \square$$

Combination C3 is to be preferred to combinations C1 and C2 for the following reasons.

First, in C1 and C2, in order to check whether or not the data has been mutilated, in C1 and C2, both the (confidential) data and the hash value should be decrypted, whereas, in C3, it suffices to decrypt the hash value.

Moreover, for this purpose, in C1 and C2, knowledge of both $A_{(u)}^{(-1)}$ and $P_{(u)}$ is required, whereas, in C3, knowledge of $A_{(u)}^{(-1)}$ suffices. If an asymmetric cryptosystem is used for signing and verification, C3 has the advantage over C1 and C2 that any user can perform the check (and thus, the check operation can be delegated to others), since $A_{(u)}^{(-1)}$ is publicly known, whereas, in C1 and C2, the check can only be performed by authorized users that know $P_{(u)}$.

Thus, regardless whether a symmetric or asymmetric cryptosystem is used for signing and verification, combination C3 is to be preferred to combination C1 and C2, and therefore, combination C3 will be used to protect data against unauthorized reading and undetected mutilation by others.

4.1.3. Fault-tolerant and secure data storage

In order to make a data storage system both fault-tolerant and secure, the techniques described in Section 4.1.1. and 4.1.2. respectively, should be combined. For any user *u*, any alternative consists of a combination (in any order) of the following three techniques:

- ❏ encryption of the original data (by means of applying function $P_{(u)}^{(-1)}$)
- ❏ adding an encrypted hash value to the data (i.e., signing the data) (by means of applying function $A_{(u)}$)
- ❏ splitting the data in a fault-tolerant way into a number of data fragments (by means of applying N different functions S_i ($1 \leq i \leq N$), that yield the N different data fragments that will be stored on N different processors)

We only consider alternatives in which data is encrypted before it is signed. As pointed out in Section 4.1.2.4., these alternatives are to be preferred to alternatives in which data is signed before it is encrypted. Then, dependent on the order in which the three techniques are applied, three alternatives (described in Section 4.1.3.1.) are possible. The evaluation of the alternatives in Section 4.1.3.2. shows, that the third alternative is to be preferred.

In Section 4.1.3.3., it is shown that in the preferred alternative, the secret cryptographic function of the applied cryptosystem is needed in order to recover lost or corrupted data fragments. In Section 4.1.3.4., it is shown that the responsibility of encryption and decryption of data with the secret cryptographic function of any user cannot be given to a single processor, but should be given to a **group** of processors instead. In order to achieve this goal, so-called ***distributed cryptographic function application protocols*** (or DCFAPs) can be used. These protocols will be introduced in Section 4.1.3.4. In Section 4.1.3.5., a possible implementation of the recovery process (i.e., the process that is responsible for recovery of lost or corrupted data fragments) is given. This implementation makes use of a so-called waiting-and-synchronization process. Section 4.1.3.6. described a possible implementation of this waiting-and-synchronization process.

4.1.3.1. Different solutions for fault-tolerant and secure data storage

In order to create fault-tolerant and secure data storage, the following solutions are proposed:

Alternative 1
In this alternative, first, the original data is encrypted. Then, the encrypted data is signed. Finally, the signed and encrypted data is split in a fault-tolerant way into a collection of data fragments. See Figure 4.7.

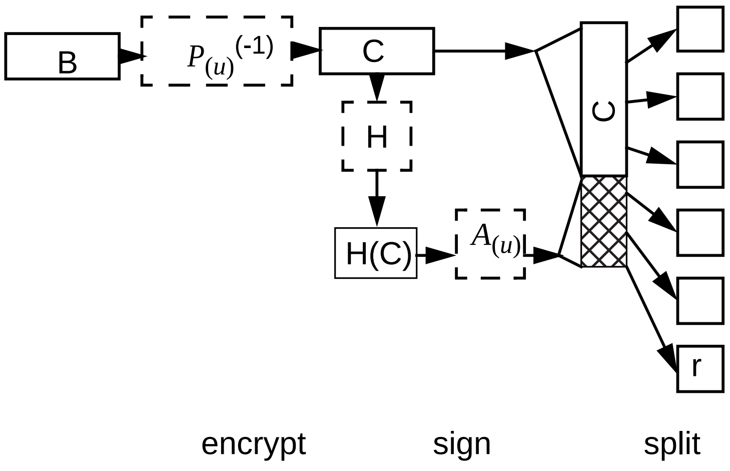


Figure 4.7. Alternative 1 (encrypt -> sign -> split)

Thus, any user u, who wants to store a piece of data, B, in a fault-tolerant and secure way, first encrypts the data, then signs it. The resulting encrypted and signed data, R, is defined by

$$R = P_{(u)}^{(-1)}(B), A_{(u)}(H(P_{(u)}^{(-1)}(B)))$$

Next, the encrypted data, R, is split in a fault-tolerant way into a collection of N constituent data fragments R_i ($1 \le i \le N$). For any i ($1 \le i \le N$), data fragment R_i is obtained by applying function S_i on data R. Hence

$$R_i = S_i(R) = S_i(P_{(u)}^{(-1)}(B), A_{(u)}(H(P_{(u)}^{(-1)}(B)))) \quad ❑$$

Alternative 2
In this alternative, the original data is first split in a fault-tolerant way into a collection of data fragments. Then, every data fragment is encrypted and signed. See Figure 4.8.

Thus, any user u, who wants to store a piece of data, B, in a fault-tolerant and secure way, first splits B in a fault-tolerant way into a collection of N constituent data fragments B_i ($1 \le i \le N$), with $B_i = S_i(B)$. Next, every data fragment B_i is encrypted and signed. The resulting encrypted and signed data fragment R_i ($1 \le i \le N$) is defined by

$$R_i = P_{(u)}^{(-1)}(S_i(B)), A_{(u)}(H(P_{(u)}^{(-1)}(S_i(B)))) \quad ❑$$

Alternative 3
In this alternative, first, the original data, B, is encrypted. The resulting encrypted data, C, is split in a fault-tolerant way into a collection of data fragments $C_1 \ldots C_N$. Finally, the encrypted data fragments are signed. See Figure 4.9.

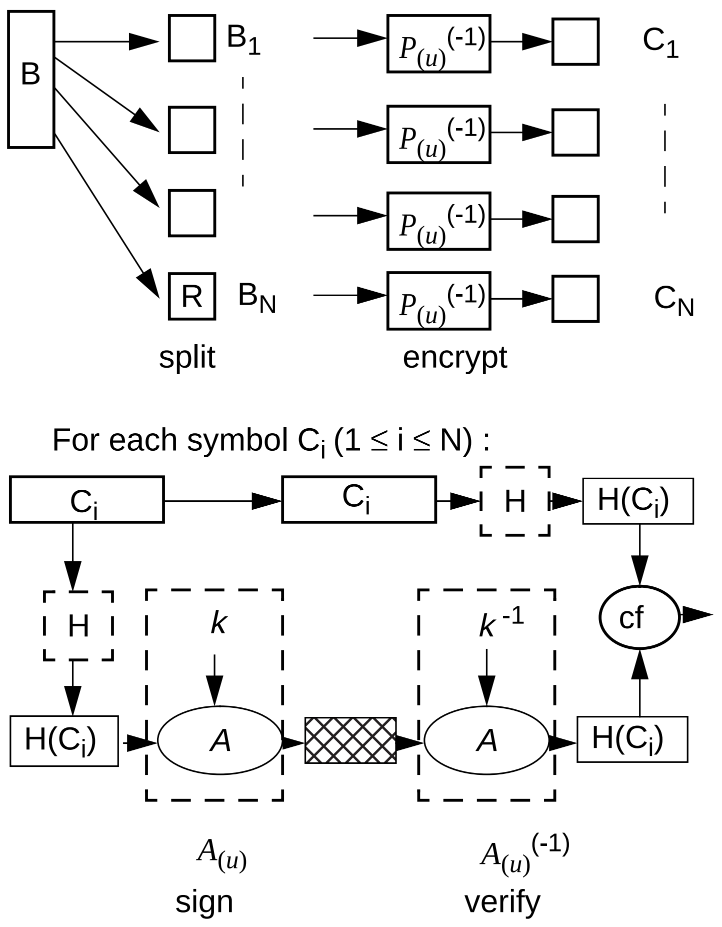


Figure 4.8. *Alternative 2 (split -> encrypt -> sign)*

The encrypted data C is equal to $P_{(u)}^{(-1)}(B)$. Hence, for all i (with $1 \le i \le N$), $C_i = S_i(P_{(u)}^{(-1)}(B))$. The encrypted and signed data fragments R_i ($1 \le i \le N$) are defined by

$$R_i = C_i, A_{(u)}(H(C_i)) = S_i(P_{(u)}^{(-1)}(B)), A_{(u)}(H(\ S_i(P_{(u)}^{(-1)}(B)))) \qquad ❏$$

4.1.3.2. Evaluation of the proposed alternatives for fault-tolerant and secure data storage

The three alternatives mentioned in the previous section are different, unless the method of data replication (see Section 4.1.1.) is applied to split the data into data fragments.

Assume instead that the more sophisticated method described in Section 4.1.1. is used, in which the data is split into a collection of data fragments by encoding the data into symbols of an error-correcting code[6]. Then, alternative 2 and 3 are to be preferred to alternative 1, since, as will be shown below, in alternative 2 and 3, in contrast to alternative 1, it is possible to encode the data into symbols of a so-called ***erasure-correcting code*** (e.g., a Reed-Solomon code).

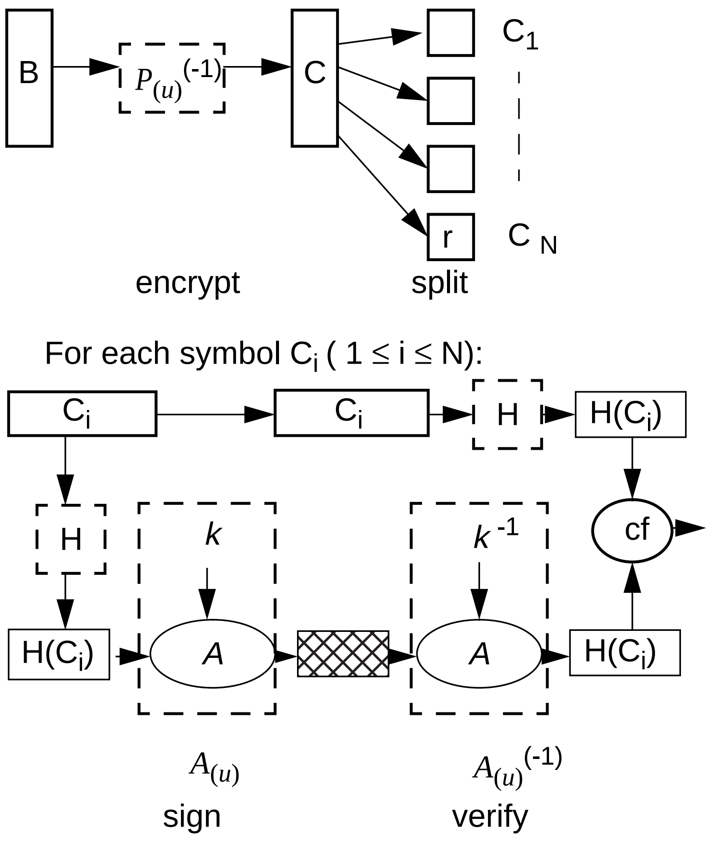


Figure 4.9. Alternative 3 (encrypt -> split -> sign)

An erasure-correcting code is a special kind of error-correcting code, capable of correcting a number of ***erasures*** (i.e., errors of which the position in the code word is known). A ***T-erasure-correcting code*** is an erasure-correcting code, capable of correcting up to T erasures.

The advantage of applying an erasure-correcting code for encoding of data over applying a regular error-correcting code, is that, in order to tolerate a certain fixed maximum number, T, of mutilated symbols in a code word, for $T > 0$, an erasure-correcting code requires much less redundancy than an error-correcting code[7]. Moreover, for $T \geq 2$, an erasure-correcting code is easier to decode than an error-correcting code [MaSl78].

A necessary requirement for a T-erasure-correcting code to be capable of correcting up to T mutilated symbols, is that the locations of the mutilated symbols in a code word can always be determined.

6. It is assumed that the applied error-correcting code is not a simple ***repetition code*** (i.e., an error-correcting code in which all symbols are identical and equal to the original data), since then, applying an error-correcting code to encode data boils down to the method of data replication.

In alternative 2 and 3, every separate encrypted and signed data fragment R_i ($1 \leq i \leq N$) contains a signed hash value, by means of which it is possible to detect any mutilations of R_i (See also Section 4.1.2.4.). In alternative 1, it may be impossible to determine per data fragment whether it has been mutilated or not, since redundant information is not available per fragment.

In alternative 2 and 3, in order to prevent malicious processors from undetectably replacing a fragment R_i with a syntactically correct old fragment R_i', or a fragment obtained from a colluding malicious processor, every fragment R_i should be unique, i.e., it should contain a unique identification (e.g., a unique version number and a unique symbol number). By including such a unique identification in the signed hash value of each data fragment, the locations of all mutilated symbols in a code word can always be detected.

Because, in alternative 2 and 3, it is possible to apply an erasure-correcting code for encoding data into symbols (something which is impossible in alternative 1), alternative 2 and 3 are to be preferred to alternative 1. In the sequel, we will assume that the data is encoded into symbols of a T-erasure-correcting code.

We will now show that alternative 3 is to be preferred to alternative 2.

In the proposed fault-tolerant and secure data storage system, arbitrary failures of a number of processors should not lead to permanent loss or corruption of data. If, due to one or more processor failures, one or more data fragments have got lost or corrupted, these data fragments should be recovered. How recovery of lost or corrupted data fragments is done, depends on which of the above-mentioned alternatives (alternative 2 or 3) is selected.

If *alternative 2* is selected, recovery of a lost or corrupted data fragment can be done as follows. We know that any original piece of data, B, has been stored in a fault-tolerant and secure way as a collection of N encrypted and signed data fragments R_i ($1 \leq i \leq N$). In this alternative, the data, B, has first been encoded into symbols of a T-erasure-correcting code, and thereafter, the data fragments, B_i ($1 \leq i \leq N$), have been encrypted. In order to recover any lost or corrupted encrypted and signed data fragment R_j (for some j, with $1 \leq j \leq N$), the *original data*, B, should first be retrieved. The original data, B, can only be retrieved with the help of the remaining data fragments R_i ($1 \leq i \leq N$, and $i \neq j$). These data fragments, however, are in encrypted form. They have to be decrypted before the original data, B, can be retrieved. Thus, in order to be able to *decrypt* the remaining encrypted data fragments, knowledge of the secret cryptographic function $P_{(u)}$ is required.

7. Let k be the number of symbols in a data word, and n be the number of symbols in a code word. Then, from [MaSl78], we know that a ***T-error-correcting code*** (i.e., an error-correcting code in which a maximum of T mutilated symbols can always be corrected) can be constructed if and only if $n \geq k + 2T$. From [VaOo89], we know that a ***T-erasure-correcting code*** (i.e., an erasure-correcting code in which a maximum of T mutilated symbols can always be corrected) can be constructed if and only if $n \geq k + T$. From [Krol91], we know that the above codes always exist if the symbol size b satisfies $b \geq {}^2\log(n\text{-}1)$. In literature, these codes are known as ***Maximum Distance Separable codes*** (or MDS codes).

If *alternative 3* is selected, then recovery of a lost or corrupted data fragment can be done without knowledge of $P_{(u)}$. In this alternative, any piece of data, B, has been stored in a fault-tolerant and secure way as a collection of N encrypted and signed data fragments R_i ($1 \leq i \leq N$), defined by $R_i = C_i, A_{(u)}(H(C_i))$. The data, B, has first been encrypted, resulting in C, and after that, the encrypted data, C, has been encoded into symbols of a T-erasure-correcting code. In order to recover any lost or corrupted encrypted and signed data fragment R_j (for some j, with $1 \leq j \leq N$), it suffices to retrieve the encrypted data, C. This encrypted data, C, needs not to be decrypted in order to be able to recover the lost or corrupted data fragments. This can be seen as follows.

In order to retrieve the encrypted data, C, the encrypted hash value of any fragment R_i should be decrypted with $A_{(u)}^{(-1)}$. If the result after decryption is equal to $H(C_i)$, then C_i was uncorrupted. If the number of corrupted symbols is less than or equal to T, then C can be recovered (since we assumed the data has been encoded into symbols of a T-erasure-correcting code). Re-creation of the lost or corrupted data fragments can be done by splitting C in a fault-tolerant way into a collection of data fragments, and signing these data fragments [8].

Thus, in alternative 3, it is possible to delegate the recovery operation to third parties without having to supply them $P_{(u)}$, i.e., the secret cryptographic function that is required to decrypt and read the data. If alternative 2 would be selected, then for recovery, $P_{(u)}$ must always be supplied. So, alternative 3 is to be preferred to alternative 2.

Therefore, we assume that alternative 3 is applied, and that the data is encoded into symbols of a T-erasure-correcting code.

4.1.3.3. Recovery of lost or corrupted data fragments requires knowledge of the secret cryptographic function of the cryptosystem applied for signing and verification

In the proposed fault-tolerant and secure data storage system, arbitrary failures of a number of processors should not lead to permanent loss or corruption of data. If, due to one or more processor failures, one or more data fragments have got lost or corrupted, it is desirable that these data fragments are recovered as soon as possible (See also Section 4.1.1.). A possible implementation of a ***recovery service*** (i.e., a service that provides for recovery of lost or corrupted data fragments) is given in Section 4.1.3.5.

According to alternative 3 in Section 4.1.3.1., any piece of data, B, has been stored in a fault-tolerant and secure way as a collection of N encrypted and signed data fragments R_i ($1 \leq i \leq N$), defined by $R_i = C_i, A_{(u)}(H(C_i))$. Recovery of any lost or corrupted encrypted and signed data fragment R_j (for some j, with $1 \leq j \leq N$) is done by re-creating this fragment.

This means that the creation of the data fragment should be redone. Recreation of lost or corrupted data fragments consists of the following steps:

8. Regardless whether a symmetric or an asymmetric cryptosystem is applied, in order to be able to re-create the recovered data fragments again, the secret cryptographic function $A_{(u)}$ is needed to sign the data fragments (i.e., add an encrypted hash value to each fragment).

❑ retrieval of the encrypted data, C, from the set of remaining encrypted and signed data fragments.
❑ splitting the encrypted data, C, again into the same[9] collection of N encrypted data fragments C_i $(1 \leq i \leq N)$
❑ signing the fragments C_i that were lost or corrupted.

Since the encrypted data, C, has been encoded into symbols of an erasure-correcting code, retrieval of data C is possible provided that the locations of all lost or corrupted data fragments are known. For this purpose, the correctness of any fragment R_i should be verified. This can be done by decrypting the hash value of any fragment R_i, by applying $A_{(u)}^{(-1)}$ to the encrypted hash value of R_i.

If a symmetric cryptosystem is used for signing and verification, then $A_{(u)}^{(-1)}$ is a secret cryptographic function. If an asymmetric cryptosystem is used for signing and verification, then $A_{(u)}^{(-1)}$ is publicly known.

The secret cryptographic function $A_{(u)}$ is needed to sign the recovered encrypted data fragments C_i (i.e., add an encrypted hash value to each fragment).

Thus, it holds that in order to recover lost or corrupted data fragments, the cryptographic functions $A_{(u)}$ and $A_{(u)}^{(-1)}$, applied for signing and verification, are needed. At least one of these functions is a secret cryptographic function.

Several solutions to the problem of recovery of lost or corrupted data fragments will be proposed in Section 4.2. and 4.3. The details of these solutions can be found in these sections. A common characteristic of each of these solutions, however, is that function $A_{(u)}^{(-1)}$ must be publicly known, in order to enable every processor to verify whether or not a piece of data has correctly been signed with cryptographic function $A_{(u)}$. As a consequence, an *asymmetric* cryptosystem is required for signing and verification.

Hence, in order to recover lost or corrupted data fragments of a user u, application of the secret cryptographic function $A_{(u)}$ is required in order to sign the re-created recovered data fragments.

4.1.3.4. Distributed cryptographic function application protocols

Since, in general, the secret cryptographic function required to recover the data fragments is only known to the owner of the data, it may seem straightforward to let the owner of the data be responsible for the recovery of lost data fragments of his/her data. However, this would imply that *the owner's processor must function correctly in order to make recovery possible*.

In other words, it would mean that the capability of the system to recover lost or corrupted data fragments would depend on the correctness of a single processor (viz. the owner's processor). This is unacceptable, since the system should be resilient to a

9. See also Section 4.1.3.5.

number of failures of *arbitrary* processors. In general, it is impossible to let a single processor be responsible for signing data fragments with a secret cryptographic function, since a failure of this processor may either inhibit recovery of lost or corrupted data fragments, or enable undetected mutilation or unauthorized reading (in case the processor is compromised). Therefore, the responsibility of signing data fragments with the secret cryptographic function of any user should be given to a **group** of processors.

Assume that a group, N, of processors is responsible for application of a secret cryptographic function F on a piece of data. Up to T processors in N may behave maliciously. In the presence of up to T faulty processors in N, it should be possible for the correct processors in N to sign data fragments with cryptographic function F. Furthermore, a group of up to T faulty processors in N may not be capable of unauthorized reading (and undetected mutilation, respectively) of data encrypted with $F^{(-1)}$ (and F respectively).

The above-mentioned problem can be solved by means of a so-called ***distributed cryptographic function application protocol*** (or ***DCFAP*** for short). Such a protocol is executed by a set, N, of N processors p_i ($1 \le i \le N$), up to T of which may behave maliciously, in order to apply a secret cryptographic function F on a certain piece of data, B, while satisfying the following properties:

Pr1. malicious behaviour of up to T processors from N does not inhibit application of cryptographic function F.

Pr2. any group of N-T or more processors is capable of applying function F.

Pr3. T or fewer colluding processors are unable to compute the secret cryptographic function F, or to apply F without the help of one or more correct processors.

A DCFAP will be defined such that application of a DCFAP guarantees that all correct processors in N reach agreement on the result of application of function F on the commonly known piece of data, B. Two different types of DCFAPs will be presented in Section 4.2. and 4.3., respectively.

It is assumed that all correct processors in N have already reached agreement about data B **before** the start of the DCFAP. Data B is equal to the hash value of the re-created recovered data fragment that should be signed. Agreement about data B can easily be established by having all correct processors actually calculate data B, independently of each other. For any i, with $1 \le i \le N$, let $B(p_i)$ be the data on which function F should be applied according to processor p_i.

A ***distributed cryptographic function application protocol*** (or ***DCFAP***) has similarities with an interactive consistency algorithm (ICA) (See Section 3.1.2.). The difference is that, while in a set, N, of N processors, up to T of which may behave maliciously, an ICA guarantees that all correct processors reach agreement on the set of initial values held by the N processors, in a DCFAP, the initial values of all correct processors are assumed to be equal (viz., the correct processors have a priori reached agreement about the data on which function F should be applied), and the correct processors reach agreement on the result of encryption of the common initial value held by the correct processors. Both in a DCFAP and in an ICA messages are communicated between processors. While in an ICA, messages are simply distributed from all processors to all processors, in a DCFAP, messages are not only distributed but also encrypted.

Just like in an ICA, the design of a DCFAP depends on the assumptions made for the system. An important parameter is the synchrony of the system. In order to make the DCFAPs as widely applicable as possible (at least, to make them applicable in a distributed system with uncertain message delay times and processor clocks running at differing rates), the DCFAPs will be based on the least restrictive ICAs with regard to the synchronicity of the system, being the authenticated self-synchronizing ICAs introduced in Section 3.4. This implies that several assumptions made for the authenticated self-synchronizing ICAs in Section 3.4. must also hold for the DCFAPs presented in Section 4.2. and 4.3.

The DCFAPs will be based on the *non-optimized* authenticated self-synchronizing ICAs presented in Section 3.4.1. It requires only minor adaptations to base the DCFAPs presented in Section 4.3. on the *optimized* authenticated self-synchronizing ICAs in Section 3.4.2.[10] The verification of this is left to the reader.

A DCFAP can be defined as follows.

Let N be a set of N processors p_i ($1 \leq i \leq N$). Every processor p_i possesses an initial value $B(p_i)$ (being the data that should be encrypted with a secret cryptographic function, say F, according to processor p_i). The initial values are encrypted and distributed from all processors to all processors by means of a DCFAP. In the presence of up to T maliciously behaving processors, a DCFAP guarantees satisfaction of the following two conditions:

DCFAP1. All correct processors agree on the result of application of cryptographic function F on the initial values they think they have received from each of the processors.

DCFAP2. For every processor p_i ($1 \leq i \leq N$), it holds that if p_i is correct, the above-mentioned agreement equals the result of application of cryptographic function F on the initial value actually possessed by processor p_i.

A DCFAP is *fault-tolerant* in such a way that properties Pr1 and Pr2 (given at the beginning of this section) are fulfilled. Furthermore, a DCFAP is *secure* in such a way that property Pr3 is fulfilled.

Notice that, for $N \leq 2T$, properties Pr2 and Pr3 conflict. However, in both solutions of

10. In contrast to the DCFAPs presented in *Section 4.3.*, it is, in general, not straightforward to base the DCFAPs presented in *Section 4.2.* on the optimized self-synchronizing ICAs, since, in the latter DCFAPs, the subsets of processors to which a received message is relayed in each communication phase of the multicast process of the DCFAP should not only contain at least one correct processor, but it should also be guaranteed that, after the multicast process of the DCFAP, the message has been signed with all component cryptographic functions (See Section 4.2.1.) that together make up function F. In Section 4.2.5., we will consider several ways in which the DCFAPs presented in Section 4.2. that are based on the second cck-distribution can be optimized w.r.t. the required communication overhead during the multicast process.

DCFAPs described in the next two sections, it holds that $N \geq 2T+1$. In the solution in Section 4.3., $N \geq 2T+1$ is satisfied by assumption. It will now be shown that for the solution in Section 4.2., it also holds that $N \geq 2T+1$. The solution in Section 4.2. makes use of some so-called ***component cryptographic key (cck) distribution*** (see Section 4.2.3. for details). Different cck-distributions are possible. In Section 4.2.3., two different possible cck-distributions are proposed; one of them requires the system to satisfy $N \geq 2T+1$, the other one requires the system to satisfy $N \geq T^2+T+1$. For any nonnegative integer value T, it holds that $T^2+T+1 \geq 2T+1$. Since T is a nonnegative integer value representing the maximum number of faulty processors that is tolerated, in the DCFAPs presented in Section 4.2., it holds that $N \geq 2T+1$, and hence, properties Pr2 and Pr3 do not conflict.

From properties Pr2 and Pr3 respectively, it can be concluded that there exists a certain minimal number of processors that is both necessary and sufficient in order to be able to apply a cryptographic function F on a piece of data. Thus, somehow, the secret information needed to apply the secret cryptographic function F must be distributed among the processors in N, such that **cooperation** of a certain minimal number of processors is both necessary and sufficient to apply the secret cryptographic function F on a piece of data. This is achieved by giving every processor in N the capability to perform part of the application of function F.

4.1.3.5. A possible implementation of the recovery service

In this section, we will describe a possible implementation of the recovery service in a synchronous system (with arbitrary nonnegative ρ and τ_{max}, and arbitrary nonnegative $\tau_{min} \leq \tau_{max}$) consisting of a set N, of N processors p_i (with $1 \leq i \leq N$), up to T of which are allowed to behave maliciously. It is assumed that the recovery service is periodically executed, and that it is implemented as a set of N replicated processes in an N-modular redundant system. See Figure 4.10. From Figure 4.10., it can be easily seen that, besides the recovery process, the processors in the N-modular redundant system may also execute other processes (indicated in Figure 4.10. as process 1,…, process k).

We assume that the same set of processors is used both to execute the recovery service and to store all data in a fault-tolerant and secure way. Every file, B, in the system is stored as a set of data fragments, R_i, on the disks of the processors $p_i \in N$. For any i, with $1 \leq i \leq N$, processor p_i stores data fragment R_i. The recovery service is responsible for recovery of data fragments that have got lost or corrupted.

The recovery process consists of several steps, which will be specified below. Various steps of the recovery process require changes to the data stored on the processors in N. Execution of such a step changes the so-called ***state*** of the system. All state-changing steps in the recovery process are designed such that it is guaranteed that the correct processors agree on the new state of the system as soon as they all have finished the state-changing step (provided that the system contains T or fewer faulty processors). However, due to differences in protocol synchronization between correct processors, in general, the correct processors will not finish a common state-changing step simultaneously.

Hence, after having finished such a state-changing step, any correct processor in the

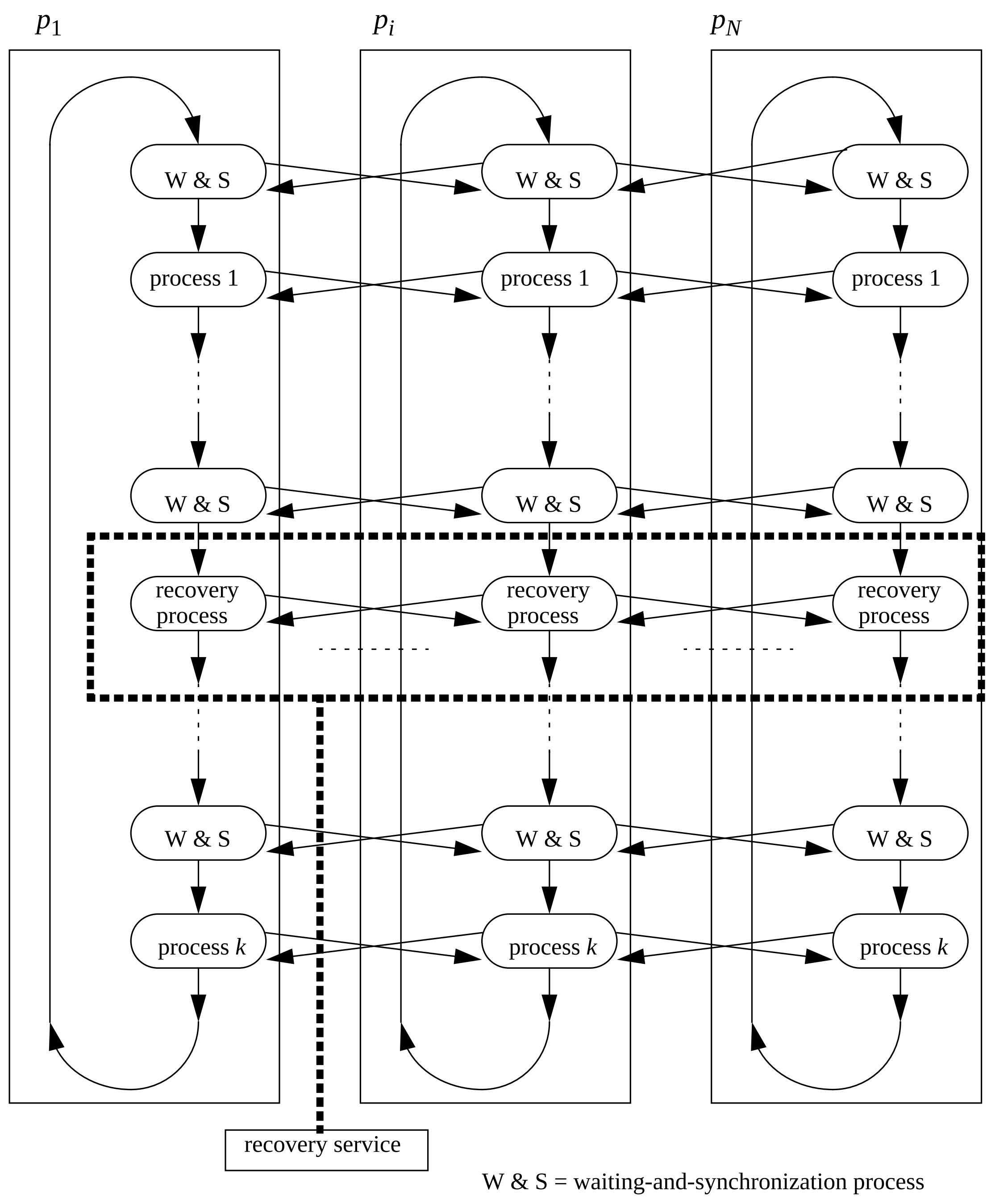


Figure 4.10. Implementation of the recovery service as a set of N replicated processes in an N-modular redundant system.

system must wait until all correct processors have also finished this state-changing step, before it can proceed with the next step of the recovery process, in order to avoid inconsistency among the correct processors. This waiting is performed in a so-called ***waiting-and-synchronization process*** (discussed in detail in the next section). Besides having the correct processor perform a waiting period, execution of a waiting-and-synchronization process also establishes approximate synchronization between the correct processors (within a real-time bound of τ_{max}, as will be shown in the next section), in order to avoid subsequent steps in the recovery process to be dependent of differences in protocol synchronization that have come into existence in or before the preceding state-changing step.

It is assumed that all correct processors start the recovery process at approximately the same time. The required protocol synchronization can be established by having all correct processors execute a waiting-and-synchronization process prior to the execution of the recovery process.

Assume that a piece of data, B, has been split into a number of encrypted and signed data fragments R_i ($1 \le i \le N$). Fragment R_i is stored on the disk of processor p_i. Assume that up to T fragments R_i have got lost or corrupted. Recovery of lost or corrupted data fragments is done by having every correct processor execute its recovery process (belonging to the recovery service).

In its recovery process, each correct processor $p_i \in N$ executes the following steps (See also Figure 4.11):

1. In the first step, processor p_i initiates an authenticated self-synchronizing Byzantine Agreement Protocol (See Chapter 3 for details) in order to distribute the signed and encrypted data fragment R_i stored on its disk to all other processors in N. Since every correct processor $p_j \in N$ will distribute its signed and encrypted data fragment R_j, in any case, p_i will receive all fragments stored on the disks of the correct processors.

2. Then, processor p_i decrypts the hash value of every remaining signed and encrypted data fragment that it has received in the previous step. Because an *asymmetric* cryptosystem is used for signing and verification of data fragments, processor p_i can decrypt the hash values of any correct data fragments received in step 1, since the cryptographic function needed to decrypt these hash values, is publicly available.

3. Provided that p_i has received a sufficient number of correct data fragments, p_i is able to correctly reconstruct the original data, B. After reconstruction, the data B can again be encoded into symbols of an erasure-correcting code. It is assumed that the data is encoded in exactly the same way as it had been encoded before recovery (i.e., the same erasure-correcting code is selected, resulting in the same set of symbols (i.e. data fragments)). As a consequence, only the data fragments that had got lost or corrupted need to be signed.

4. Since step 3 is a state-changing step, the correct processors should first run a waiting-and-synchronization process, before proceeding with the next step.

5. Finally, all newly created recovered data fragments should be signed. For this purpose, processor p_i starts a DCFAP for any data fragment that must be signed. Notice that, since all correct processors will start a DCFAP for every data fragment, every correct processor will receive a signed version of every data fragment at least N-T times.

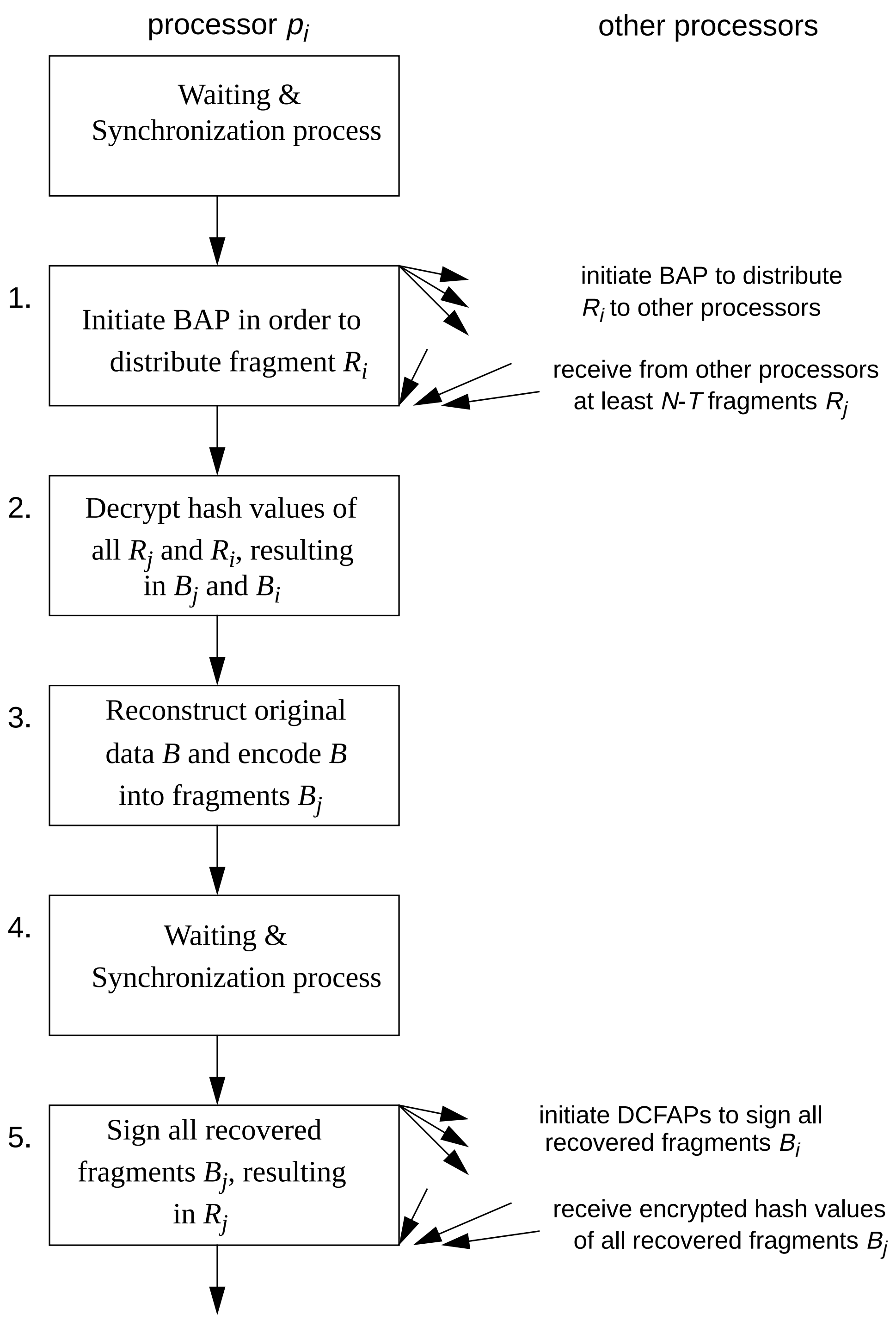

processor p_i
other processors
Waiting & Synchronization process
1.
Initiate BAP in order to distribute fragment R_i
initiate BAP to distribute R_i to other processors
receive from other processors at least N-T fragments R_j
2.
Decrypt hash values of all R_j and R_i, resulting in B_j and B_i
3.
Reconstruct original data B and encode B into fragments B_j
4.
Waiting & Synchronization process
5.
Sign all recovered fragments B_j, resulting in R_j
initiate DCFAPs to sign all recovered fragments B_i
receive encrypted hash values of all recovered fragments B_j

Figure 4.11. The recovery process executed in a correct processor p_i, which is the k-th processor in the order to initiate a DCFAP to sign recovered data fragments

4.1.3.6. A possible implementation of the waiting-and-synchronization process

In this section, a possible implementation of the waiting-and-synchronization process mentioned in the previous section, will be presented. The process is replicated on a set, N , of N processors in a synchronous system (with arbitrary nonnegative ρ and τ_{max}, and arbitrary nonnegative $\tau_{min} \leq \tau_{max}$). It is assumed that up to T processors in the system may behave maliciously.

A waiting-and-synchronization process is executed in order to avoid inconsistency between correct processors after execution of a state-changing step, and to establish tight protocol synchronization between the correct processors in the system. The waiting-and-synchronization process is designed such that the process does not terminate before all correct processors have terminated the preceding state-changing step, and, as a result of execution of the waiting-and-synchronization process, the maximum real-time difference in protocol synchronization between all correct processors is brought down to τ_{max}. The process starts from the assumption that, prior to its execution, the maximum real-time protocol synchronization between the correct processors in the system is equal to some nonnegative value D [11].

This assumption is justified, provided that a bounded known maximum real-time difference in protocol synchronization between the correct processors can be achieved as a result of execution of the preceding process(es). This condition can be met by selecting a *self-synchronizing* protocol to precede the waiting-and-synchronization process.

We will now start by giving an informal description of the waiting-and-synchronization process. Thereafter, a formal description of the process is given. The general idea of the waiting-and-synchronization process for the case T=1 has been depicted in Figure 4.12. As will be explained below, during the process, several messages may be exchanged. In Figure 4.12., only those messages that are accepted have been depicted.

Informally, the waiting-and-synchronization process can be described as follows. Assume that the maximum real-time difference in protocol synchronization between the correct processors in the system is equal to D. Then, every correct processor $c \in N$ starts the waiting-and-synchronization process by waiting until it is sure that a time interval with a real-time length of at least D has passed, or (as will be explained later in this section) as soon as it has terminated the preceding state-changing step and it has received claims from a sufficient number of processors that they have terminated their waiting period. After having waited this long, c is sure that all correct processors have ended the preceding process, and hence, all correct processors might, in fact, go on with the next process (be it that, prior to proceeding with the next process, the maximum real-time difference in protocol synchronization between the correct processors should first be brought down to τ_{max}).

As long as processor c has not received claims from a sufficient number of processors that they have terminated their waiting period, c will wait until a real-time period of length of at least D has passed. Since c's processor clock may run up to a factor $(1+\rho)$

11. Notice that this assumption implies that, after a real-time interval of length D, all correct processors have terminated the state-changing step that precedes the waiting-and-synchronization process.

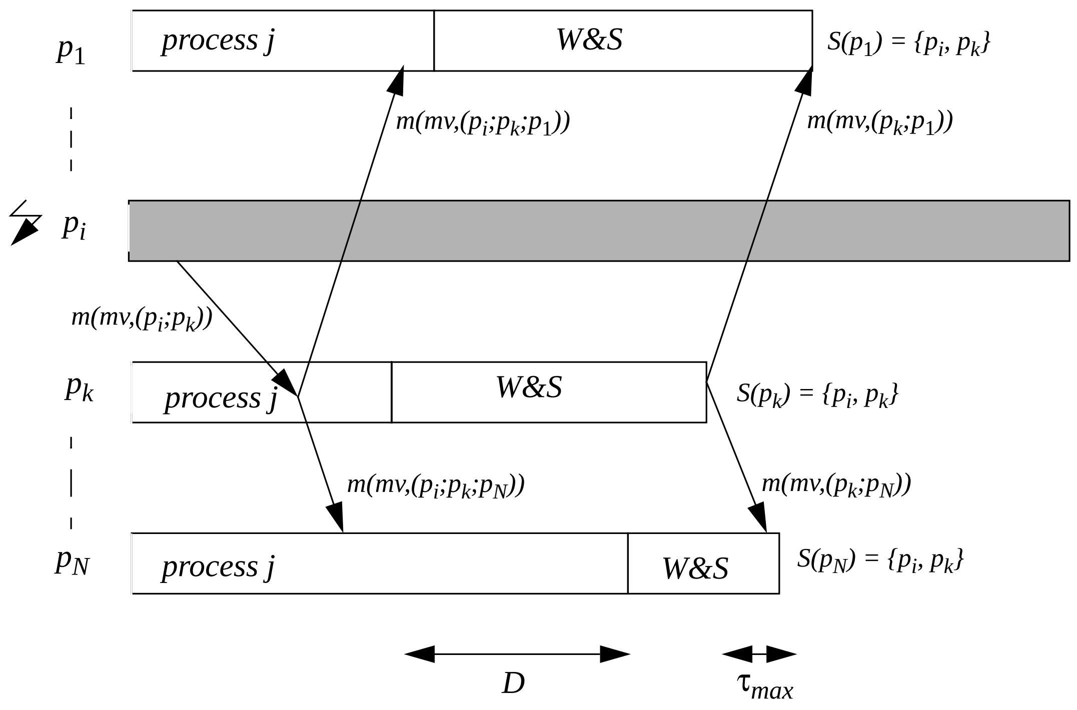


Figure 4.12. General idea of the waiting-and-synchronization (W&S) process for the case T=1 (with faulty processor p_i)

faster than real-time, *c* has to wait until a time interval with a length of $D{\cdot}(1{+}\rho)$, measured by *c*'s processor clock, has passed. After *c* has waited $D{\cdot}(1{+}\rho)$, measured by its own processor clock, it informs every other processor $j \in N$ of the fact that it has waited a time interval with real-time length of at least *D* by sending *j* a valid message[12] $m(mv,(c;j))$. The message value *mv* of this message is assumed to uniquely identify the waiting-and-synchronization process that the message belongs to. When processor *j* receives a message $m(mv,(c;j))$, processor *j* is informed that the source processor of this message (or more precisely[13], the processor of which its identification is the first one in the path information of this message), in this case, processor *c*, claims to have ended its waiting period.

In general, processor *c* does **not** conclude the waiting-and-synchronization process after having informed all other processors of having reached the end of its waiting period. As will be explained below, processor *c* concludes the waiting-and-synchronization process only as soon as *c* has received claims from at least *T*+1 different processors (including *c* itself iff *c* has ended its waiting period) that they have ended their

12. The definition of a valid message and the message syntax of valid messages is the same as in Section 3.3.
13. Notice that, if the processor of which the identification is the first one in the path information of the received message *m*, say processor *p*, is faulty, then it is possible that another faulty processor, *q*, has used *p*'s signature, and created message *m* with path information starting with the identification of *p*. Strictly speaking, processor *q* is the source processor for message *m*. However, if, in this section, we speak of the source processor of a valid message *m* , we always mean the processor of which the identification is the first one in the path information of *m*, unless explicitly stated otherwise.

waiting period.

While processor c is waiting until a time interval with real-time length of at least D has passed, it is possible that c receives valid messages originating from other processors in the system, indicating that they have ended their waiting period. As will be explained below, processor c maintains a set $S(c)$ of processors that have claimed to c to have ended their waiting period. Initially, this set $S(c)$ is empty. Assume that processor c receives a valid message $m(mv,(s;\underline{\mathbf{p}};c))$ while it is waiting until the time interval with real-time length of at least D has passed. Then processor c concludes that processor s has ended its waiting period. If c did not receive a valid message with source processor s before, c adds s to $S(c)$ (the set of processors that did report to c that they have ended their waiting period). Provided that $(s;\underline{\mathbf{p}};c)$ contains fewer than T+2 processors, processor c signs message $m(mv,(s;\underline{\mathbf{p}};c))$ and relays it to every processor $h \in N$ that is not in $(s;\underline{\mathbf{p}};c)$. The reason why this is done is given below. Processor c will end its waiting period if c has waited $D\cdot(1+\rho)$ (measured by its own processor clock), or as soon as it knows that at least one correct processor has ended its waiting period.

Since the system may contain up to T faulty processors, which may try to mislead correct processors by informing one or more of them that they have ended their waiting period, processor c can only conclude that at least one correct processor has ended its waiting period, as soon as it has received a valid message from at least T+1 different processors. This conclusion is based on the valid messages that c has received during its waiting period. Since these valid messages may originate from faulty processors, and hence, may not have been sent to all other correct processors, processor c makes sure that all correct processors that did not yet receive the message will receive it from c, by relaying any valid message it receives (and which is sent by a source from which c did not receive a valid message before) to all processors, the identification of which is absent in the path information of the received message. In this way it is ensured that, as soon as one correct processor has received valid messages originating from at least T+1 different processors, within a real-time interval of at most τ_{max} (i.e., the maximum real-time interval needed to communicate any valid message, hence, also the last valid message), every correct processor has received a valid message from at least T+1 different processors. By having the correct processors end the waiting-and-synchronization process as soon as they have received valid messages originating from at least T+1 different processors, the maximal real-time difference in protocol synchronization between the correct processors at the end of the waiting-and-synchronization process is equal to τ_{max}. The proof of this is straightforward and left to the reader.

The waiting-and-synchronization process can be formally described as follows.
Initially, for each processor $i \in N$, $S(i) = \emptyset$. Processor i starts the waiting-and-synchronization process as soon as it has concluded the preceding process.

For each correct processor $i \in N$:

Event: *Since the start of the waiting-and-synchronization process, processor i has waited a time interval of length $D\cdot(1+\rho)$, as measured by its own processor clock.*

Action: 1. *Processor i sends a valid message $m(mv,(i;j))$ to every processor $j \in N \setminus \{i\}$.*

2. *Processor i concludes its waiting period and adds its own identification to S(i), i.e., S(i) := S(i) $\cup$ {i}. If S(i) contains at least T+1 elements, then processor i concludes its waiting-and-synchronization process.*

Event: *message mess$_j$(i) (with $1 \leq j \leq M_i$) received*

Action:

1. *Check if mess$_j$(i) is a valid message, i.e., check if mess$_j$(i) $\in$ ValidMessages*
   If invalid(mess$_j$(i)) then reject mess$_j$(i) and abort, i.e., do no perform further actions on mess$_j$(i).
   *If valid(mess$_j$(i)), then mess$_j$(i) = m(mv,(**q**)) for some path (**q**) $\in$ ValidPaths and some mv $\in$ MessageValues. If the message value mv of m(mv,(**q**)) indicates a waiting-and-synchronization process that processor i has already concluded, then reject m(mv,(q)) and abort. If last((**q**)) $\neq$ i, then reject mess$_j$(i) and abort, otherwise mess$_j$(i) = m(mv,(**p**;i)) for path (**p**) $\in$ ValidPaths with (**p**) = prefix((**q**)) and length((**p**)) $\geq$ 1.*
2. *Check if a valid message mess$_k$(i) (with $1 \leq k < j$) with first(path(mess$_k$(i))) = s (with s = first((**p**))) has already been accepted.*
   *If such a message has already been accepted, then reject mess$_j$(i) and abort, otherwise, accept m(mv,(**p**;i)).*
3. *If i accepted m(mv,(**p**;i)) and path(m(mv,(**p**;i))) contains fewer than T+2 processors, for every processor h that is not in path(m(mv,(**p**;i))), i signs message m(mv,(**p**;i)) and i relays m(mv,(**p**;i)) to h, immediately after m(mv,(**p**;i)) was received by i. As stated in Section 3.3.3.3., relaying message m(mv,(**p**;i)) to any processor h results in m(mv,(**p**;i;h)) being sent to h.*
4. *Add s to S(i), i.e., S(i) := S(i) $\cup$ {s}. If S(i) contains at least T+1 elements, then processor i concludes its waiting-and-synchronization process.*

All correct processors are assumed to execute the above-described protocol. The behaviour of faulty processors, however, may deviate arbitrarily from the above-described protocol.

4.1.4. Related work

In literature, several related problems have been investigated. In this section, these related problems will be briefly discussed, and the difference with the DCFAPs proposed in this chapter will be indicated.

In Section 4.1.4.1., the notion of so-called ***multisignature schemes*** [Okam88] will be discussed. Section 4.1.4.2. investigates the so-called ***secret-sharing schemes*** or ***threshold schemes***, originally proposed by Blakley in [Blak79] and Shamir in [Sham79], respectively. Furthermore, in Section 4.1.4.3., ***threshold cryptography*** will be discussed.

4.1.4.1. Multisignature schemes

In order to delegate a sign operation to a group of processors, in [Okam88], the use of ***multisignature schemes*** was proposed. Roughly speaking, a multisignature scheme is a scheme in which any data is subsequently signed by a group of N processors[14]. The processors may sign in any order[15]. However, all processors must sign the data, so the multisignature scheme in [Okam88] is not fault-tolerant. In Section 4.2., we propose a new, fault-tolerant variant of the multisignature scheme in [Okam88] that can be used to implement a distributed cryptographic function application protocol.

The multisignature scheme presented in [Okam88] consists of three phases: the key generation and publication phase, the multisignature generation phase, and the multisignature verification phase. These phases are described below. See also Figure 4.13.

In the *key generation and publication phase* (or *KGP*), every processor p_i ($1 \le i \le N$) that joins in the sign operation generates a one-way hash function H_i, and a public and secret cryptographic function ($K_{(i)}^{(-1)}$ and $K_{(i)}$, respectively) of a bijective public-key cryptosystem. The hash function H_i and the public cryptographic function $K_{(i)}^{(-1)}$ are published.

In the *multisignature generation phase* (or *MGP*), a distinction is made between the signature generation of the first signing processor, and the signature generation of the j-th signing processor (with $j > 1$). The first signer, say p_i, generates a message m, and adds an encrypted hash value $K_{(i)}(H_i(m))$ to this message. Processor p_i sends M_1, S_1 (where $M_1 = m$, and $S_1 = K_{(i)}(H_i(m))$) along with his identifier ID_i to the second signer. The j-th signer ($2 \le j \le N$), say p_k, receives a message M_{j-1}, S_{j-1} that has been signed by (j-1) other processors. Processor p_k verifies this message in the multisignature verification phase, described below. If message is valid, then p_k signs it, and, if $j < N$, it sends M_j,S_j (where $M_j = M_{j-1}$, and $S_j = K_{(k)}(S_{j-1})$) along with (ID_i, …, ID_k) to the (j+1)-th signer, otherwise, M_j,S_j along with (ID_i, …, ID_k) is sent to a signature verifier.

In the *multisignature verification phase* (or *MVP*), the signature verifier verifies a received message M_j,S_j (for some j, with $1 \le j \le N$) by using the public cryptographic functions $K_{(i)}$ ($1 \le i \le N$) in two steps as follows. In the first step of the multisignature verification phase, in a recursive way, message M_1',S_1' is obtained. The verification starts with $M_j' = M_j$, and $S_j' = S_j$. For any k, with $2 \le k \le j$, message M_{k-1}',S_{k-1}' is obtained from message M_k',S_k' as follows. Let ID_h (for some h, with $1 \le h \le N$) be the

14. The multisignature scheme presented in [Okam88] discusses some details, e.g., addition of padding characters when the length of the signature (i.e., hash value) is incompatible with the blocklength required by the applied cryptographic functions). Since, in this section, we are only interested in the general idea, such details are omitted from the above description of multisignature schemes.
15. Notice that having a number of processors generate a multisignature on a piece of data is different from having a number of processors sign a message in an authenticated self-synchronizing BAP (presented in Section 3.3.), since, in contrast to multisignature schemes, in an authenticated self-synchronizing BAP, the order in which the message is signed is essential and can be determined from the message.

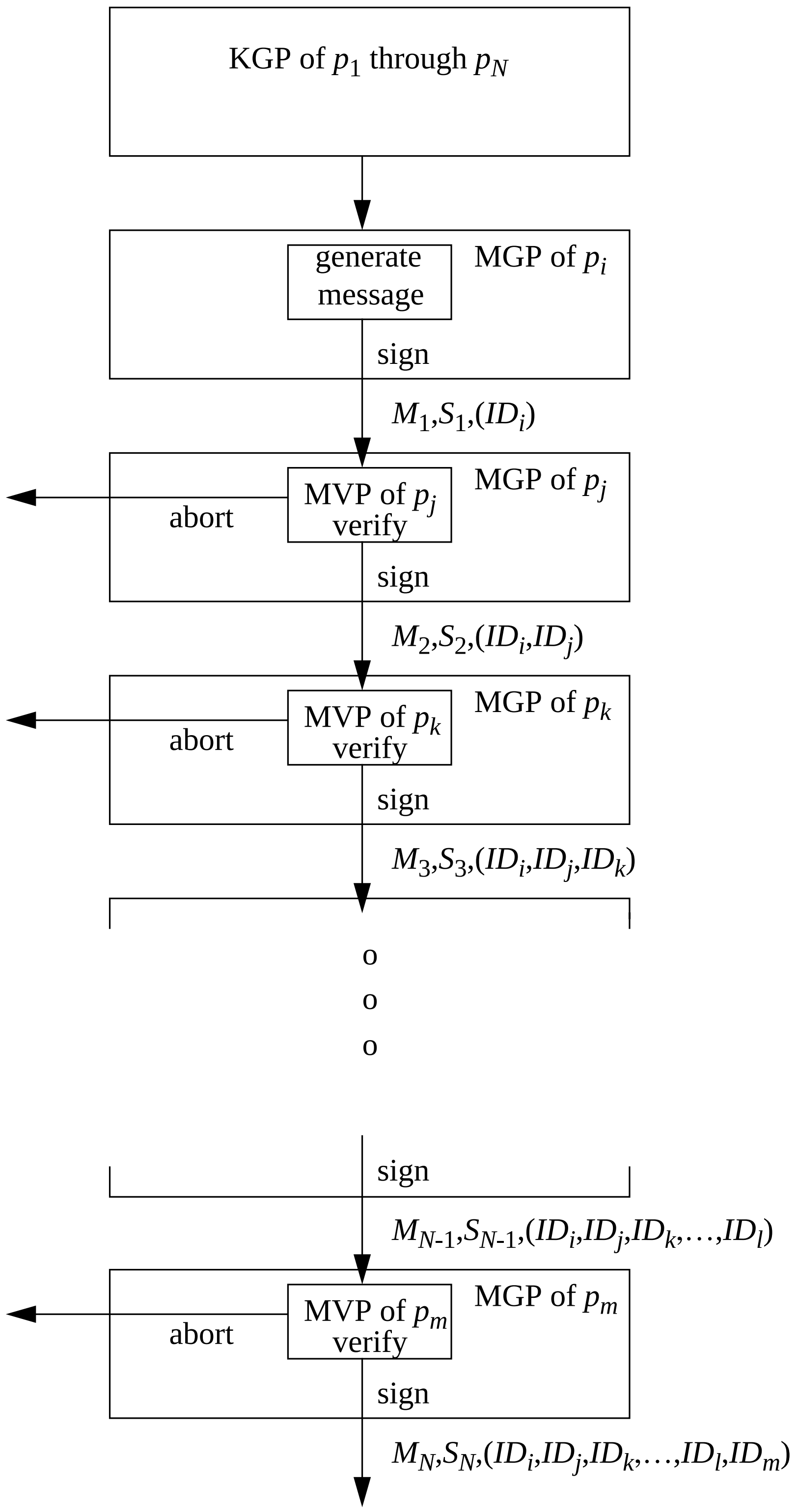


Figure 4.13. A multisignature scheme

k-th processor identification in the list of identifiers. Then, if message M_j',S_j' is valid, processor p_h is the k-th processor that has signed the message, we calculate $M_{k\text{-}1}' = M_k'$, and $S_{k\text{-}1}' = K_{(h)}^{(-1)}(S_k')$. In the second step, the validity of message M_1',S_1' is checked. The received message M_j, S_j is considered valid, if M_1', S_1' obtained in step 1, satisfies $K_{(g)}^{(-1)}(S_1') = H_g(M_1')$, where ID_g (for some g, with $1 \leq g \leq N$) is the first processor identification in the list of identifiers.

Okamoto states (in [Okam88]) that the multisignature scheme can be considered as safe as the public-key cryptosystem used in the multisignature scheme. However, the security of the multisignature scheme relies on several very subtle assumptions not mentioned in the paper. A careful design of the multisignature scheme is needed to avoid so-called ***protocol failures***. In [Simm92, p.544], a protocol failure is defined as an instance in which the protocol fails to provide the advertised level of security or authentication, not due to the cryptosystem that is used, but due to the design of the protocol. E.g., choosing RSA as the public-key cryptosystem, and having every processor in the multisignature scheme base its public and secret cryptographic function on a common modulus, makes the multisignature scheme vulnerable to a ***common modulus protocol failure***. The details of this type of protocol failure can be found in [Simm92, p.546-549]. An overview of protocol failures in cryptosystems is given in [Simm92, p.543-558].

4.1.4.2. Secret sharing schemes

In 1979, Blakley [Blak79] and Shamir [Sham79] independently developed ***secret sharing schemes*** (or ***threshold schemes***) in order to enable robust key management.

In a secret sharing scheme, the power to regenerate a secret cryptographic key is shared. The information needed to construct a secret cryptographic key K is distributed among a group of processors as a set of N related pieces of information (called ***shadows*** [Blak79], or ***shares*** [Sham79]), K_1, …, K_N, such that any set of T+1 or more of these shadows will suffice to recover the original cryptographic key, but such that no subset of T or fewer shadows will reveal any information about the cryptographic key. Each processor is given a number of shadows[16]. For simplicity, unless stated otherwise, we assume that each processor is given exactly one shadow. In [Simm92, p.446], a ***perfect secret sharing scheme*** is defined as a secret sharing scheme in which insiders (i.e., processors possessing one or more shadows) and all groups of unauthorized insiders have no advantage over outsiders (i.e., processors not possessing shadows) in guessing the secret. In the Byzantine fault model, it is desirable that any applied secret sharing scheme is perfect, since no individual processor (either a processor possessing one or more shadows, or a processor not possessing any shadows) can be unconditionally trusted.

The secret sharing schemes in [Blak79] are based on projective geometry, whereas the schemes in [Sham79] are based on Lagrange interpolation (see, e.g., [ArAs70, CoBo72]) of a polynomial defined over a finite field. In [Simm92, pp.445-450], it is shown that the secret sharing scheme proposed by Shamir (in [Sham79]) is perfect, while the secret sharing scheme proposed by Blakley (in [Blak79]) isn't. Below, we will give a somewhat more detailed explanation of the secret sharing scheme proposed by Shamir in [Sham79]. For an extensive discussion about secret sharing schemes, the

16. Sometimes it may be desirable for processors to have differing capabilities to recover the secret. In this case, Shamir, in [Sham79] suggests to give more than one shadow to the more capable processors.

reader is referred to [Simm92, pp.441-497].

The following description of the secret sharing scheme proposed by Shamir in [Sham79] is obtained from [Simm92, p.445-446].

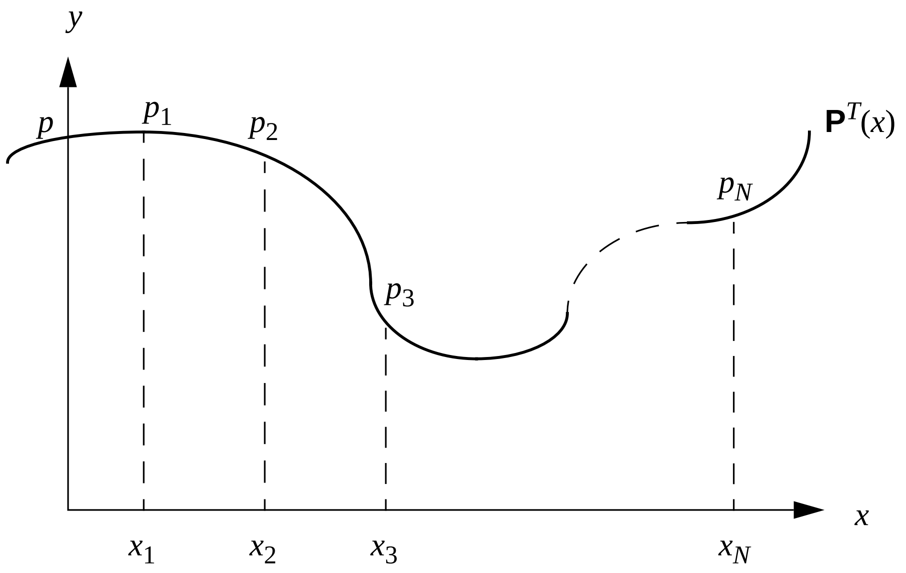


Figure 4.14. Shamir's secret sharing scheme

Shamir's secret sharing scheme is based on interpolation of a polynomial defined over a finite field, GF(q). Figure 4.14. illustrates the basic idea. Given (T+1) distinct points in the two-dimensional plane, (x_i, y_i), for $1 \le i \le T+1$, there is a unique T-th degree polynomial, $\mathbf{P}^T(x)$, for which $\mathbf{P}^T(x_i) = y_i$, for all i with $1 \le i \le T+1$. If the secret cryptographic key is taken to be an element $p \in$ GF(q), it can be partitioned into N shadows as follows. A T-th degree polynomial $\mathbf{P}^T(x)$ is chosen randomly. $\mathbf{P}^T(x)$ has the representation

$$\mathbf{P}^T(x) = \sum_{i=0}^{T} a_i x^i$$

where the constant term a_0 is the secret cryptographic key p; i.e., $\mathbf{P}^T(0) = a_0 = p$. The N shadows, $p_i = (x_i, y_i)$, with $N \ge T$, are obtained by evaluating $\mathbf{P}^T(x)$ at N distinct points, x_i, with $x_i \ne 0$, i.e., $p_i = y_i = \mathbf{P}^T(x_i)$.

Given any subset of (T+1) of these points, it is computationally easy using Lagrange interpolation [ArAs70, CoBo72] to calculate a_0 (i.e., to recover the secret cryptographic key p, since $a_0 = p$). Given only k (with $k \le T$) points on $\mathbf{P}^T(x)$, a_0 could equally likely be any element in GF(q), since for each choice of a point p' on the y-axis there will be equally many T-th degree polynomials through the subset of k points and p'. Consequently, any collusion of fewer than (T+1) of the processors that possess a shadow will have no better chance of determining the secret cryptographic key than will a processor that possesses no shadows at all. To all groups of processors that possess fewer than (T+1) of the shadows, the secret cryptographic key is equally likely to be the y-coordinate of any point on the line (0, y), i.e., the probability of guessing p will be 1/q. Thus, Shamir's secret sharing scheme is perfect in the sense described above.

Because (T+1) shadows are required to reconstruct the cryptographic key K, exposure of up to T shadows does not endanger key K, and no group of processors that possesses fewer than (T+1) of the shadows can collude to obtain the key. At the same time, loss of shadows does not inhibit recovery of key K, provided that at least (T+1) valid shadows are available.

Since at least (T+1) shadows are needed in order to recover the original cryptographic key, a group of processors possessing at least (T+1) shadows should cooperate to recover key K. Moreover, a set of at least (T+1) shadows should be given to **one single trusted processor** in order to give this processor the capability to recover the cryptographic key.

In our distributed cryptographic function application protocols, we do not want to rely on the trustworthiness of a single processor, since the system should be resilient to a number of arbitrary failures of arbitrary processors. Thus, for our purpose, the method of secret sharing is not directly applicable. In the next section, however, we will see that the principle of secret sharing can be used in an approach to share the power to apply a secret cryptographic function. In this approach it is not necessary to rely on the trustworthiness of a single processor. This approach, called ***function sharing*** [DDFY94], can be used to obtain distributed cryptographic function application protocols, as will be shown in Section 4.3.

4.1.4.3. Threshold cryptography

Instead of sharing the information needed to generate a secret cryptographic key, as it is done in the method of secret sharing described above, in ***threshold cryptography***, the capability of applying a secret cryptographic function is shared. In [DDFY94] the term ***function sharing*** is introduced for this purpose.

The first threshold cryptosystems were independently developed by Boyd in [Boyd89], Croft and Harris in [CrHa89] and Desmedt in [Desm88]. Since then, a vast amount of literature has appeared on this subject. For an overview of threshold cryptography, the reader is referred to [Desm93, Desm94].

The basic idea of function sharing (or threshold cryptography) is to split a trapdoor one-way function F into N so-called ***shadow functions*** (or ***share-functions***) F_i ($1 \le i \le N$), that are distributed among N processors p_i ($1 \le i \le N$), up to T of which may behave maliciously. Each processor p_i is given one shadow function. The intractable function F becomes easy to compute at a given point when given any threshold (at least (T+1) out of N) of correct evaluations of different shadow functions F_i at that point. Otherwise, function F remains hard to compute. Furthermore, function F should remain intractable even after exposing up to T shadow functions F_i and exposing values of all shadow functions at polynomially many inputs. In many solutions to the problem of function sharing, it is assumed that all processors p_i apply their shadow function on a commonly available piece of input data, B.

Most of the research in threshold cryptography aims at designing ***non-interactive solutions***, i.e., solutions in which the processors may calculate their share (i.e., the result of application of their shadow function F_i on the commonly available input data B) independently of the other processors, in other words, no interaction between the processors is needed to be able to successfully apply the shadow function F_i. Usually, the results of the processors are all sent to a single so-called ***combiner***, that combines the received shares in order to calculate the result of application of the secret cryptographic function F on the commonly available input data B.

The idea of function sharing is illustrated by Figure 4.15.

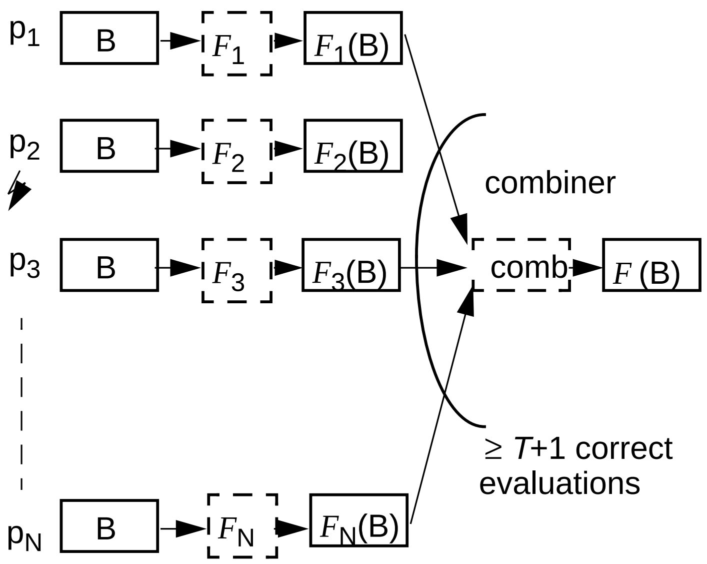


Figure 4.15. Function sharing

Several implementations of function sharing satisfying the above requirements, mostly based on a combination of a secret sharing technique with an asymmetric cryptosystem have appeared in literature. For an overview, see, e.g., [Desm94]. In [FrDe92], a solution is presented, which is based on a combination of RSA and the secret sharing scheme proposed by Shamir (in [Sham79]), and which is proven to be as secure as RSA.

In most implementations of function sharing, a correct result is only guaranteed provided that the shareholders do not jam the computation (e.g., by outputting random data). I.e., if one or more of the shareholders produce random data instead of their share, the combiner will combine the results of the shareholders into an incorrect result.

So, in order to obtain a correct result, the combiner should combine T+1 correct results. In a system consisting of at least 2T+1 processors, up to T of which may be faulty, the combiner will receive T+1 or more shares. In general, it is impossible for the combiner to determine which shares are correct and which are not. The combiner knows, however, that at least one combination of T+1 of the received shares yields a correct result.

If the combiner would have a possibility to verify whether an obtained result is correct, the combiner would be able to produce a result for any combination of T+1 shares, and select one of the correct results at random. Provided that the combiner has the disposal over the original data B, and the inverse function[17] $F^{(-1)}$ of the secret cryptographic function F, the combiner is able to verify the correctness of the result of the combination of T+1 shares. This is explained as follows. Let the result of some combination of T+1 shares obtained by the combiner be X. Then, the combiner decides X to be correct iff $F^{(-1)}(X) = B$.

In a DCFAP, the technique of function sharing can be applied with that difference, that the results of application of the shadow functions generated by the processors p_i ($1 \le i \le N$) should be sent to a number of combiners (viz. all processors p_i themselves should act as combiners) instead of to one single combiner. In Section 4.3., it is described how the above-described technique of function sharing can be used in combination with an interactive consistency algorithm to implement a DCFAP. See Figure 4.16.

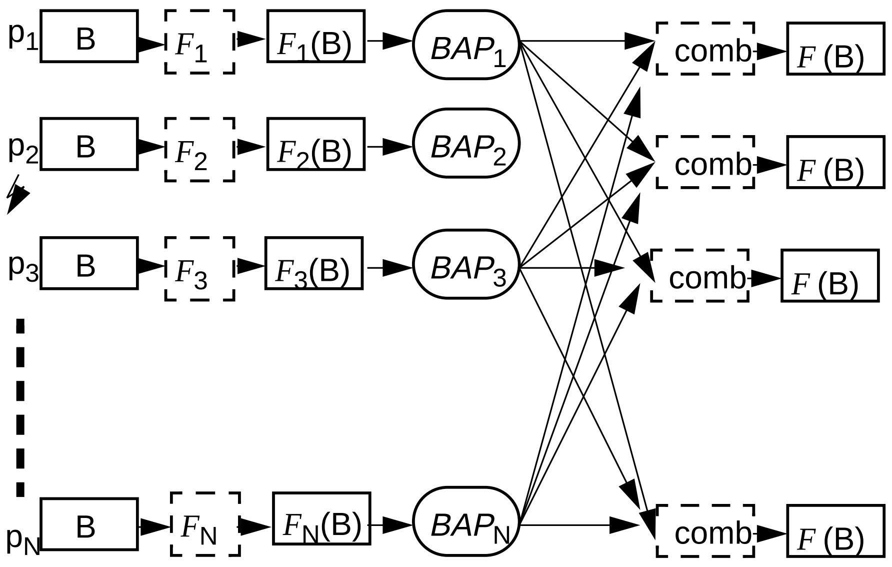


Figure 4.16. A DCFAP based on function sharing

4.1.5. Overview

In the next sections we will introduce two different types of distributed cryptographic function application protocols. As already stated in Section 4.1.3.4., these protocols are executed by a group, N, of N processors p_i ($1 \le i \le N$), up to T of which may behave maliciously, in order to apply a secret cryptographic function F on a piece of data, in

17. This requirement can be satisfied by letting F and $F^{(-1)}$ be functions of an asymmetric cryptosystem. Then, function $F^{(-1)}$ can be made public, while function F is kept secret.

such a way that the following requirements are fulfilled:

Pr1. malicious behaviour of up to *T* processors from *N* does not inhibit application of cryptographic function *F*.
Pr2. any group of *N*-*T* or more processors is capable of applying function *F*.
Pr3. *T* or fewer colluding processors from *N* are unable to compute the secret cryptographic function *F*, or to apply *F* without the help of one or more correct processors.

In Section 4.2., we present an ***interactive DCFAP*** (i.e., a DCFAP in which the processors interact in order to calculate their share) based on a fault-tolerant version of the multisignature scheme in [Okam88]. In Section 4.3., a ***non-interactive DCFAP*** (i.e., a DCFAP in which the processors may calculate their share independently of the other processors) is presented, based on the principle of function sharing in threshold cryptography. The non-interactive DCFAP is to be preferred to the interactive DCFAP, since, in the non-interactive DCFAP, the minimally required number of processors is smaller than in the interactive DCFAP, and, in general, the non-interactive DCFAP requires much less computation overhead than the interactive DCFAP. The DCFAPs based on function sharing, which are presented in Section 4.3., have appeared previously in [PoKM97].

4.2. Distributed cryptographic function application protocols based on a fault-tolerant multisignature scheme

In this section, distributed cryptographic function application protocols (DCFAPs) are introduced, which are based on a fault-tolerant version of the multisignature scheme in [Okam88] (discussed in Section 4.1.4.1.).

In these DCFAPs, it is assumed that any secret cryptographic function can only be constructed as a composition of a number of so-called ***component cryptographic functions***. The secret cryptographic key corresponding to a component cryptographic function of a cryptographic function *F* is called a ***component cryptographic key*** of *F*. Component cryptographic functions and component cryptographic keys are the topics of Sections 4.2.1. and 4.2.2. respectively.

As will be shown in Section 4.2.3., the distribution of the component cryptographic keys over the set of processors can be done in various ways. In this section, two possible component key (cck) distributions are proposed. The first cck-distribution is optimal with regard to the minimal number of processors needed, however, this cck-distribution requires many component cryptographic keys. The second cck-distribution requires fewer component cryptographic keys, this cck-distribution is not optimal with regard to the required number of processors and the amount of data communication, however.

In Section 4.2.4., we investigate how the actual cryptographic function application takes place. Finally, in Section 4.2.5., for a DCFAP based on the second cck-distribution, we describe some ways to reduce the execution time of the DCFAP and the required amount of data communication needed in the DCFAP.

4.2.1. Component cryptographic functions

As stated before, in any DCFAP, each processor is given the capability to perform part of a secret cryptographic function. The cryptographic function can be completed by having a sufficiently large set of processors subsequently apply their part of the cryptographic function.

For this purpose, every processor $p_i \in N$ ($1 \le i \le N$) generates N public and N secret cryptographic functions, $K_{(i,j)}^{(-1)}$ and $K_{(i,j)}$ respectively, of a bijective public-key cryptosystem. The public cryptographic functions $K_{(i,j)}^{(-1)}$ are published.

For any piece of data, B, the secret cryptographic function $K_{(i)}$ of processor p_i is defined by:

$$K_{(i)}(B) = K_{(i,1)}(K_{(i,2)}(\ldots K_{(i,N)}(B)\ldots))$$

The cryptographic functions $K_{(i,j)}$ ($1 \le j \le N$) are called the ***component cryptographic functions*** of $K_{(i)}$. We assume that these component cryptographic functions satisfy the following assumptions:

CCF1. For any piece of data B, it holds that:

$$K_{(i,1)}(\, K_{(i,2)}(\, \ldots\, K_{(i,N)}(B)\, \ldots\,)) = K_{(i)}(B) \qquad (\textit{compositionality})$$

CCF2. For any piece of data B, and any component cryptographic functions $K_{(i,j)}$ and $K_{(i,k)}$ ($j \neq k$), it holds that:

$$K_{(i,j)}(K_{(i,k)}(B)) = K_{(i,k)}(K_{(i,j)}(B)) \qquad (\textit{commutativity})$$

CCF3. $(\forall\, j,k \in [1, N] :: j \neq k \Rightarrow K_{(i,j)} \neq K_{(i,k)})$

CCF4. All component cryptographic functions ($K_{(i,1)}$ through $K_{(i,N)}$) of $K_{(i)}$ are needed in order to be able to apply function $K_{(i)}$.

CCF5. It is computationally infeasible to calculate any $K_{(i,j)}$ ($1 \le j \le N$) from any subset of $\{K_{(i,k)} \mid 1 \le k \le N \wedge k \neq j\}$, i.e., the set of all component cryptographic functions of $K_{(i)}$ except $K_{(i,j)}$.

Encryption of a piece of data, B, with the secret cryptographic function $K_{(i)}$ of processor p_i is done by subsequently applying all secret component cryptographic functions $K_{(i,j)}$ ($1 \le j \le N$) on that piece of data, B. By CCF2, the component cryptographic functions may be applied in any order.

By giving every processor $p_j \in N$ the capability to apply some of the component cryptographic functions $K_{(i,j)}$, it can be guaranteed that the cryptographic function $K_{(i)}$ can only be applied by having a sufficiently large set of processors subsequently apply (all or some of) their component cryptographic functions.

Just like in the multisignature scheme in [Okam88], careful design of the DCFAP is needed to avoid protocol failures. E.g., choosing RSA as the public-key cryptosystem, and having every processor in the DCFAP generate public and secret cryptographic function pairs based on a common modulus, as it is done in [Boer96, PBHS96], makes the DCFAP vulnerable to the common modulus protocol failure. See also Section

4.1.4.1.

4.2.2. Component cryptographic keys

Application of a component cryptographic function $K_{(i,j)}$ of a bijective public-key cryptosystem boils down to execution of a commonly known algorithm, of which the output does not only depend on the input data, but also on a cryptographic key $k_{i,j}$ fed into the algorithm. Thus, the cryptographic key $k_{i,j}$ (called a ***component cryptographic key***) determines the output of component cryptographic function $K_{(i,j)}$. As a consequence, for any component cryptographic function to be a secret function, only the applied component cryptographic key needs to be kept secret.

We will assume that for every processor $p_i \in N$, the component cryptographic keys $k_{i,j}^{(-1)}$ corresponding to p_i's public component cryptographic functions, $K_{(i,j)}^{(-1)}$ are published, whereas the component cryptographic keys $k_{i,j}$ corresponding to p_i's secret component cryptographic functions $K_{(i,j)}$ are given to authorized processors only.

Assume that the result of application of a certain secret cryptographic function F depends on a secret cryptographic key f. Assume that functions F_1 through F_N are the component cryptographic functions of function F, and that $f_1 \dots f_N$ are the corresponding secret component cryptographic keys of F. Now, for any $i \in [1, N]$, applying the component cryptographic function F_i on a piece of data, B, boils down to applying the commonly known cryptographic algorithm of the applied cryptosystem with component cryptographic key f_i and the piece of data, B, as inputs.

In this chapter, the following notations will be used:

N4.1. As stated before, a secret cryptographic function consists of a commonly known algorithm and a secret cryptographic key supplied to it. For any secret cryptographic function F, the corresponding commonly known algorithm and secret cryptographic key are denoted by F and f respectively.

N4.2. For any secret cryptographic function F, and any piece of data B, application of function F on B consists of applying the corresponding commonly known algorithm F with cryptographic key f on data B. The result will be denoted by $F(f,B)$ or by $F(B)$.

The assumptions CCF1 through CCF5 that are made for component cryptographic functions F_1 through F_N of function F (with corresponding cryptographic key f), can be satisfied if the corresponding component cryptographic keys $f_1 \dots f_N$ of F satisfy the following assumptions:

K1. For any piece of data B, it holds that:

$$F(f_1, (F(f_2, (\dots F(f_N, B) \dots)) = F(f, B) \qquad (\textit{compositionality})$$

K2. For any piece of data B, and any two component cryptographic keys f_i and f_j ($i \neq j$), it holds that:

$$F(f_i, F(f_j, B)) = F(f_j, F(f_i, B)) \qquad (\textit{commutativity})$$

K3. $(\forall\, i,j \in [1, N] :: i \neq j \Rightarrow f_i \neq f_j)$

K4. All component cryptographic keys (f_1 through f_N) of cryptographic function

F are needed in order to be able to apply function F.

K5. It is computationally infeasible to calculate any component cryptographic key f_i from any subset of $\{f_j \mid 1 \leq j \leq k \wedge j \neq i\}$, i.e., the set of all component cryptographic keys of cryptographic function F except component cryptographic key f_i.

In order to satisfy the properties Pr1 through Pr3 of the DCFAP stated in Section 4.1.5., the component cryptographic keys f_1 through f_N, needed to apply the secret cryptographic function F, are replicated and distributed among the N processors in N in such a way that the following requirements are fulfilled:

R4.1. Every component cryptographic key is possessed by at least T+1 processors

R4.2. Any group of N-T or more processors possesses a sufficient number of component cryptographic keys to encrypt data with the secret cryptographic function F.

R4.3. T or fewer colluding processors do not possess a sufficient number of component cryptographic keys to generate the secret cryptographic function F or to encrypt data with F.

4.2.3. Possible component cryptographic key distributions for DCFAPs based on a fault-tolerant multisignature scheme

In this section, two different component cryptographic key distributions for DCFAPs based on a fault-tolerant multisignature scheme, are proposed. The first solution, given in Section 4.2.3.1., is optimal with regard to the required number of processors, i.e., the number of processors that is required in the proposed solution is minimal. However, this solution may require cryptographic function F to be constructed as a composition of a large number of different component cryptographic functions. Moreover, a large total number of copies of component cryptographic keys may be needed in the system. The solution proposed in Section 4.2.3.2. requires more processors, but a smaller number of different component cryptographic keys and copies of component cryptographic keys than the solution in Section 4.2.3.1. In Section 4.2.3.3., both solutions are compared.

4.2.3.1. A component cryptographic key distribution which is optimal with regard to the required number of processors

In this section, a component cryptographic key distribution will be proposed, which, as will be shown below, is optimal with regard to the required number of processors.

As discussed before, a secret cryptographic function F is constructed as a composition of a number of component cryptographic functions F_i. It is assumed that **all** corresponding component cryptographic keys f_i are needed in order to be able to apply cryptographic function F.

Now, every processor is given copies of a number of different component cryptographic keys f_i of F. It seems logical to give each processor an **equal** number, say k, of copies, since any group of N -T or more processors should be equally powerful, i.e., possess at least one copy of each component cryptographic key f_i of F (by requirement R4.2.).

While R4.2. requires that any group of N-T or more processors knows all component cryptographic keys of F, R4.3. states that no group of T or fewer processors has access to all component cryptographic keys of F. For R4.2. and R4.3. to be non-conflicting, N-T should be greater than T. In other words, $N \geq 2T+1$.

Now, for $N=2T+1$, a component cryptographic key distribution is proposed which satisfies R4.1, R4.2, and R4.3. Clearly, this solution is optimal with regard to the required number of processors, N. Hence, such a solution will be referred to as an ***optimal component cryptographic key distribution***.

Before we describe how, for arbitrary $T > 0$, the component cryptographic key distribution is constructed, the proposed component cryptographic key distribution is presented for the case $T = 2$. See Figure 4.17. For T=2, the required number of processors

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
p_1	x	x	x	x	x	x				
p_2	x	x	x				x	x	x	
p_3	x			x	x		x	x		x
p_4		x		x		x	x		x	x
p_5			x		x	x		x	x	x

x = processor p_i possesses component cryptographic key f_j

Figure 4.17. Proposed component cryptographic key distribution with a minimal number of processors for the case T = 2.

p_i is $2T+1 = 5$, whereas function F is constructed as a composition of 10 component cryptographic functions with corresponding component cryptographic keys f_j. It is easy to verify that the solution satisfies requirements R4.1, R4.2, and R4.3.

The construction of the component cryptographic key distribution is based on the following arguments.

By R4.3., a group of T or fewer processors should never possess copies of all component cryptographic keys of F. In other words, for every possible group of T processors, there should exist at least one component cryptographic key that is not in possession of this group of processors. The number of different groups of T processors within a group of 2*T*+1 processors is equal to

$$\binom{2T+1}{T}$$

Since, by R4.2 and the preliminary that *N* = 2*T*+1, any group of *T*+1 (*T*+1 = *N*-*T*, since *N* = 2*T*+1) processors should possess all component cryptographic keys, the component cryptographic key that is not in possession of a certain group of *T* processors, must be in possession of each of the other processors in *N*. (This follows also from R4.1) The foregoing implies, that for every component cryptographic key of *F*, there exists exactly one group of *T* processors that does not possess this key. So the number of different component cryptographic keys of *F* is also given by

$$\binom{2T+1}{T}$$

Now, the component cryptographic key distribution is constructed such that every component cryptographic key is given to a different set of *T*+1 processors. Since *N* = 2*T*+1, this implies that, for every possible set of *T* processors, there is at least one component cryptographic key that is not in possession of this group of processors. It is now easy to verify that the proposed component cryptographic key distribution satisfies the requirements R4.1., R4.2., and R4.3.

In the proposed optimal component cryptographic key distribution, each processor is given *k* copies of component cryptographic keys, where *k* is given by

$$k = \binom{2T}{T}$$

This can be seen as follows. Since every possible group of *T*+1 processors is given one component cryptographic key that is not in possession of the other processors in *N*, and, for each processor $p \in N$, there exist *k* different groups of *T*+1 processors that *p* is part of, each processor is given *k* cryptographic keys.

Since the system consists of 2*T*+1 processors, the total number of copies of component cryptographic keys in the system is equal to

$$(2T+1) \cdot \binom{2T}{T}$$

4.2.3.2. A component cryptographic key distribution which requires a smaller number of different component cryptographic keys and a smaller number of copies of component cryptographic keys

Although the component cryptographic key distribution proposed in Section 4.2.3.1. is optimal with regard to the required number of processors in the system, it is certainly not optimal with regard to the required number of different component cryptographic keys needed to apply *F*, and with regard to the total number of copies of component cryptographic keys in the system. This will be demonstrated by the following component cryptographic key distribution for a system consisting of $T^2 + T + 1$ processors. This solution has been proposed in [Boer96, PBHS96]. In Figure 4.18., the component cryptographic key distribution is given for the case *T* = 2. In this case, the required number of processors p_i is $T^2 + T + 1 = 7$, whereas function *F* is constructed as a composition of 7 component cryptographic functions F_j with corresponding component

	f_1	f_2	f_3	f_4	f_5	f_6	f_7
p_1	x					x	x
p_2	x	x					x
p_3	x	x	x				
p_4		x	x	x			
p_5			x	x	x		
p_6				x	x	x	
p_7					x	x	x

x = processor p_i possesses component cryptographic key f_j

Figure 4.18. *An alternative component cryptographic key distribution for the case T = 2.*

cryptographic keys f_j. It is easy to verify that this solution also satisfies requirements R4.1., R4.2., and R4.3.

In general, for a system consisting of $T^2 + T + 1$ processors, many component cryptographic key distributions exist. For arbitrary nonnegative T, the proposed component cryptographic key distribution is defined by

*For any i with $1 \leq i \leq N$ (with $N = T^2 + T + 1$), for any cryptographic function F constructed as the composition of N component cryptographic functions F_y with corresponding component cryptographic keys f_y (with $1 \leq y \leq N$), processor $p_i \in N$ possesses the k (with $k = T+1$) component cryptographic keys in the set $\{f_j \mid j = 1 + ((i + N - m)$**mod** $N) \wedge 1 \leq m \leq k\}$.*

Whether this solution is optimal with regard to the required number of different component cryptographic keys of F, or optimal with regard to the total number of copies of component cryptographic keys in the system, is an open question, the answer of which is beyond the scope of this thesis.

The construction of the component cryptographic key distribution is based on the following arguments.

Again, it is assumed that each processor is given an equal number, say k, of copies of component cryptographic keys. Now, by choosing the total number of different component cryptographic keys of F to be equal to $T{\cdot}k + 1$, it is guaranteed that T or fewer processors never possess all component cryptographic keys of F, since T or fewer processors cannot possess more than $T{\cdot}k$ component cryptographic keys of F. Hence, requirement R4.3 is always fulfilled.

Requirement R4.1. is fulfilled by ensuring that every component cryptographic key f_i ($1 \le i \le N$) is possessed by T+1 different processors. For any component cryptographic key f_i, a copy of f_i is given to a group of T+1 processors. As a consequence, any group of N-T processors always possesses at least one copy of every component cryptographic key of F, hence, requirement R4.2. is always fulfilled.

The number of processors in the system, as well as the number of copies of component cryptographic keys, k, given to each processor, can be expressed as a function of T as follows.

From the above, it may be concluded that the total number of copies of component cryptographic keys in the system is equal to $(T{\cdot}k + 1){\cdot}(T{+}1)$. Since every processor is given k copies of component cryptographic keys, the number of processors $N(T,k)$ can be expressed as a function of T and k as follows:

$$\begin{aligned} N\,(T, k) &= \lceil\, ((T \cdot k + 1) \cdot (T{+}1)) \,/\, k\, \rceil \\ &= \lceil\, T\cdot (T{+}1) + (T{+}1) \,/\, k\, \rceil \end{aligned}$$

Here, for any nonnegative value x, $\lceil x \rceil$ is defined as the smallest integer value greater than or equal to x.

For fixed nonnegative integer value T and nonnegative integer k, $N(T, k)$ is minimal if k goes to infinity, i.e.:

$$\lim_{k \to \infty} N\,(T, k) \;=\; T \cdot\; (T + 1)$$

Since k can never be infinite in any practical situation, this is only a theoretical minimum of $N(T, k)$. The practical minimum of $N(T, k)$ is denoted as $min(N(T, k))$ and defined by

$$min(N(T, k)) = T\cdot (T{+}1) + 1 \text{ if } k \ge T{+}1$$

The corresponding minimum number of component cryptographic keys k given to any processor (for fixed maximal number of faulty processors, T) is denoted as $min(k)$ and defined by

$$min(k) = T{+}1.$$

In the proposed component cryptographic key distribution, k is chosen to be equal to $min(k)$ and N is selected to be equal to $min(N(T,k))$.

4.2.3.3. Comparison of the proposed component cryptographic key distributions

In this section, the two component cryptographic key (cck) distributions proposed in Section 4.2.3.1. and 4.2.3.2. will be compared. These cck-distributions will be referred to as cck-distribution 1 (Section 4.2.3.1.) and cck-distribution 2 (Section 4.2.3.2.), respectively. In Table 4.1., for some values of *T*, the two cck-distributions will be compared with regard to the following three parameters:

- ❑ the required number of processors, *N*,
- ❑ the number of copies of component cryptographic keys per processor, *k*,
- ❑ the total number of copies of component cryptographic keys, *totalcopies*,
- ❑ the total number, *totalccks*, of component cryptographic keys of *F*.

Table 4.13: Comparison of proposed component cryptographic key distributions

T	*cck-distribution*	*N*	*k*	*totalcopies*	*totalccks*
1	cck-distr. 1 cck-distr. 2	3 3	2 2	6 6	3 3
2	cck-distr. 1 cck-distr. 2	5 7	6 3	30 21	10 7
3	cck-distr. 1 cck-distr. 2	7 13	20 4	140 52	35 13
4	cck-distr. 1 cck-distr. 2	9 21	70 5	560 105	126 21

For cck-distribution 1, the parameters are given (as a function of *T*) by

$$N = 2T+1$$

$$k = \binom{2T}{T}$$

$$totalcopies = (2T+1) \cdot \binom{2T}{T}$$

$$totalccks = \binom{2T+1}{T}$$

For cck-distribution 2, the parameters are given (as a function of *T*) by

$$N = T^2 + T + 1$$

$$k = T+1$$

$$totalcopies = (T+1)\cdot(T^2 + T + 1)$$

$$totalccks = T^2 + T + 1$$

About the required number of communication phases in the DCFAP for either of the proposed cck-distributions, we remark the following. A DCFAP consists of a number of communication phases, in which the correct processors in the system relay messages they accept after having signed any accepted message with the component cryptographic functions that they possess, and which the accepted message has not been signed with before. By R4.2., any group of at least *N*-*T* processors always possesses at least one copy of each component cryptographic key, so, as will be clear from Section 4.2.4., after *N* - *T*- 1 communication phases, in each of which messages, after having been signed, are relayed to a large enough set of processors guarantees that at least one correct processor has received a message on which all component cryptographic functions of F have been applied. In the last communication phase, this message can be relayed to all other processors in the system (see also Section 4.2.4.). For cck-distribution 1, this means that T+1 communication phases are sufficient, for cck-distribution 2, T^2+1 communication phases are sufficient. In Section 4.2.5., it is proved that for cck-distribution 2, it is also always possible to find a DCFAP with only T+1 communication phases.

Which component cryptographic key distribution is preferred in an actual implementation of a DCFAP, depends on the requirements made for the system. Cck-distribution 1 may be favorable, because this cck-distribution requires fewer processors than cck-distribution 2, be it at the cost of a larger value for *totalcopies* and *totalccks*. However, for other systems, the large number of component cryptographic keys and the total number of copies of component cryptographic keys in the system may be prohibitive. In this case, cck-distribution 2 will be preferred.

In the next section, a description is given of a DCFAP based on a fault-tolerant multi-signature scheme. Both the number of processors on which the DCFAP is executed, and the number of communication phases of the DCFAP depend on the cck-distribution that is selected for the DCFAP. For the DCFAP described in the next section, the number of communication phases is selected to be equal to K, and the number of processors on which the DCFAP is executed is selected to be equal to N. Furthermore, for the selected cck-distribution, the total number of different component cryptographic keys is selected to be equal to *totalccks*, whereas the number of copies given to each processor is selected to be equal to *copiespp*.

For cck-distribution 1, the parameters N, K, *totalccks*, and *copiespp* have the following values:

$$N = 2T+1$$
$$K = T+1$$
$$totalccks = \binom{2T+1}{T}$$
$$copiespp = \binom{2T}{T}$$

For cck-distribution 2, the parameters N, K, *totalccks*, and *copiespp* have the following values:

$$N = T^2+T+1$$
$$K = T^2+1$$
$$totalccks = T^2+T+1$$
$$copiespp = T+1$$

As will be proved in Section 4.2.5., for cck-distribution 2, it is also always possible to construct DCFAPs such that $K = T$+1 suffices.

4.2.4. Distributed cryptographic function application

In this section we will describe how the actual distributed cryptographic function application takes place. In Section 4.2.4.1., we state the required assumptions. The DCFAPs presented in this section consist of execution of a number of so-called ***partial cryptographic function application protocols*** (***PCFAPs***), which will be defined and informally described below. A formal description of a PCFAP can be found in Section 4.2.4.2. Finally, Section 4.2.4.3. contains a description of a DCFAP based on a fault-tolerant multisignature scheme.

A PCFAP is defined as follows.

Let N be a set of N processors p_i ($1 \leq i \leq N$). A certain processor p_i possesses an initial value $B(p_i)$ (being the data that should be encrypted with a secret cryptographic function, say F, according to processor p_i). PCFAP$_i$ is defined as the PCFAP in which processor p_i is the source processor, i.e., the initial value $B(p_i)$ is encrypted and distributed from processor p_i to all processors by means of PCFAP$_i$. In the presence of up to T maliciously behaving processors, PCFAP$_i$ guarantees satisfaction of the following two conditions:

PCFAP1. All correct processors agree on the result of application of cryptographic function F on the initial value they think they have received from processor p_i.

PCFAP2. If processor p_i is correct, the above-mentioned agreement equals the result of application of cryptographic function F on the initial value actually possessed by processor p_i.

A partial cryptographic function application protocol has similarities with a Byzantine Agreement Protocol (See Section 3.1.2.). Both consist of a multicast process and a decision-making process. The main difference between a PCFAP and a Byzantine Agreement Protocol (or BAP for short) is that, while in the multicast process of a BAP, messages are simply distributed from the source to the destination processors, in a PCFAP, during the multicast process, messages are not only distributed but also encrypted. Furthermore, as will be explained in Section 4.2.4.3., every correct processor will relay any valid message with path information describing a path of length K to all other processors in the system. In short, the reason for this is, that these messages are fully encrypted (i.e. encrypted with secret cryptographic function F), and the decision taken in each correct processor should be based on the same set of fully encrypted messages.

A PCFAP runs on system consisting of a set, N, of N processors interconnected by a network. One of the processors is the source. The result of encryption of the initial value of the source should be distributed to all processors (the destination processors). Notice that the source is also one of the destination processors. In general, the source starts the BAP by multicasting a message to (some or all of) the destination processors. At receipt of the first valid message, a correct destination processor will start its own sub-protocol. In order to prevent confusion, we will refer to the protocol as a whole as the ***PCFAP***, and to any ‘sub-protocol’ of the PCFAP running on a certain destination processor p as the ***sub-PCFAP of processor p***.

To get an intuitive grasp of how a PCFAP works, we will now give a rather informal description of a PCFAP. In this description, several definitions will be used, that will be explained in the sequel. Notice that the description below only serves to introduce the general idea. The details are given in Section 4.2.4.1. and 4.2.4.2., respectively.

Assume that p_i is the ***source processor*** of the PCFAP. Then, $B(p_i)$ is the data fragment that should be encrypted according to processor p_i. Processor p_i wants to communicate the result of encryption of $B(p_i)$ with secret cryptographic function F to all destination processors.

Let $CCK(p_i)$ be the set of the *copiespp* component cryptographic keys of F possessed by p_i. Now, processor p_i applies cryptographic algorithm F on $B(p_i)$ with all the component cryptographic keys it possesses, resulting in a partially encrypted data fragment $crypt(B(p_i),CCK(p_i))$.

This partially encrypted data fragment is the partially encrypted message value *emv* of the message relayed to all correctly functioning processors that did not yet participate in the encryption process. Assume that p_j is a correctly functioning processor that receives a message with message value *emv* from processor p_i. Then, at receipt of this message, p_j will first check if the message value was properly encrypted. In order to prevent a group of up to T faulty processors from encrypting erroneous data with the secret cryptographic function F, each correct processor should only apply its part of the cryptographic function on a piece of (partially encrypted) data, if it is convinced that the piece of data has not been tampered with.

For this purpose, p_j checks if the message value *emv* of the received message is valid, by decrypting this partially encrypted message value. I.e., p_j performs the cryptographic algorithm F with all public component cryptographic keys $f_y^{(-1)}$ (with $1 \leq y \leq N$) that form a component cryptographic key pair with the component cryptographic keys that are possessed by p_i , i.e., processor p_j computes $reversecrypt(emv,CCK(p_i))$, where:

$$reversecrypt(emv,CCK(p_i)) = F(f_i^{(-1)}, \ldots , F(f_{N-k+i-1}^{(-1)}, emv) \ldots)$$

Processor p_j decides that the partially encrypted data fragment was properly encrypted iff

$$reversecrypt(emv,CCK(p_i)) = B(p_j)$$

since we assume that agreement about the initial data has been reached, i.e., we assume that, for any pair of correct processors p_i, $p_j \in N$, it holds that $B(p_i) = B(p_j)$.

To be able to perform this check, processor p_j must know with which component cryptographic keys the data fragment has been encrypted. Hence, we assume that p_j knows how the component cryptographic keys have been distributed among the different processors. Then, it is only required that p_j has some information about the path (i.e., the sequence of processors) along which the data fragment has travelled from the source processor to p_j. Since processor p_j knows the path information of the received message, p_j can trace with which component cryptographic keys the data fragment has been encrypted, provided that the message has travelled along a path of correct processors only.

If processor p_j decided that the partially encrypted data fragment was properly encrypted, then processor p_j believes that processor p_i has correctly encrypted data fragment *B*, and thus, p_j cooperates in the encryption process. Processor p_j now performs cryptographic algorithm *F* on the partially encrypted data fragment $crypt(B(p_i),CCK(p_i))$ that it received with all the component cryptographic keys that it possesses, and that are not possessed by processor p_i. In other words, p_j produces a partially encrypted data fragment $crypt(crypt(B(p_i), CCK(p_i)), CCK(p_j))$.

This partially encrypted data fragment is again forwarded to one or more correctly functioning processors that did not yet participate in the encryption process, etc., until the cryptographic algorithm *F* has been performed with all secret component cryptographic keys f_i $(1 \leq i \leq N)$ of function *F*.

4.2.4.1. Assumptions

A DCFAP is defined on a set, *N*, of *N* processors p_i $(1 \leq i \leq N)$, up to *T* of which may behave maliciously. In a DCFAP, it is assumed that, prior to the start of the DCFAP, all correct processors p_i in *N* have reached agreement about the data, *B*, that should be encrypted with secret cryptographic function *F*. How this agreement can be reached has been discussed in Section 4.1.3.4. For every processor p_i, let $B(p_i)$ be the data that should be encrypted according to p_i. For any *j*, with $1 \leq j \leq totalccks$, let f_j be a component cryptographic key of cryptographic function *F*. With all component cryptographic keys f_j, with $1 \leq j \leq totalccks$, function *F* can be applied.

A DCFAP consists of execution of *N* partial cryptographic function application protocols PCFAP_i $(1 \leq i \leq N)$. Processor p_i acts as the source in PCFAP_i in order to have the result of encryption of $B(p_i)$ with secret cryptographic function *F* be communicated to the other processors in *N*.

As stated in Section 4.1.3.4., in order to make the PCFAPs as widely applicable as possible, they will be based on the least restrictive deterministic BAPs with regard to the synchronicity that is required, i.e., the PCFAPs will be based on the authenticated self-synchronizing BAPs presented in Section 3.3.6. The PCFAPs are designed in such a way that the conditions PCFAP1 and PCFAP2 stated in the previous section are satisfied.

In Section 3.3., we have proved that the authenticated self-synchronizing BAPs guarantee Byzantine Agreement, i.e., satisfaction of the interactive consistency conditions IC1 and IC2. The conditions PCFAP1 and PCFAP2 are very similar to the interactive consistency conditions. As we will see, a PCFAP which satisfies conditions PCFAP1 and PCFAP2 guarantees Byzantine Agreement about the result of encryption of the initial value sent by the source with a secret cryptographic function *F*.

Furthermore, because the PCFAPs are based on the authenticated self-synchronizing BAPs in Section 3.3.6., a number of assumptions made for these self-synchronizing BAPs in Section 3.3.8. should also hold for the PCFAPs presented in this section. For the sake of completeness, these assumptions have been listed below. Notice that the assumptions A4.4. and A4.6. through A4.8 are slightly different from the assumptions made for BAPs in Section 3.3.8., since in a PCFAP, in contrast to a BAP, messages are partially encrypted while being relayed, instead of simply being relayed.

A4.1. There exist fixed known upper and lower bounds (τ_{max} and τ_{min} respectively) on the time required to communicate a message from one correct processor to another processor in the system. Furthermore, $0 \le \tau_{min} \le \tau_{max}$.

A4.2. For any processor *p* and any real time *t*, let *C*(*p, t*) be the value of *p*'s clock at time *t*. Then, for any correct processor *p* in the system, and any two points t_1 and t_2 in real-time, we assume there exists a $\rho \ge 0$ (known by all correct processors), such that:

$$\frac{1}{1+\rho} \le \frac{C\left(p, t_1\right) - C\left(p, t_2\right)}{t_1 - t_2} \le (1+\rho)$$

Thus, we assume that the processor clocks of correct processors are ρ-bounded. Notice that the value *C*(*p*,*t*) of the clock of a correct processor *p* may deviate arbitrarily from real-time *t* (Thus, arbitrary clock skew between processor clocks of the processors in the system is allowed).

A4.3. We assume the existence of a nonnegative real-time bound Δ on clock value measurement uncertainty, known by all correct processors. The first time any correct processor *p* detects that its processor clock indicates a clock value greater than or equal to a certain clock value *v*, it is assumed that *p*'s clock indicated *v* at most Δ ago.

A4.4. A processor may concurrently execute different PCFAPs. We assume that messages of different PCFAPs can be distinguished.

A4.5. A perfect communication link is available between every pair of correct processors in the system. This assumption is justified by the fact that we can model a link failure as a failure of one of its adjacent processors [SiLL90].

A4.6. Every valid message *m* contains a message value and authenticated path information. Knowledge of the path information of a valid message *m* is sufficient to group *m* in the decision-making process and to determine a set of processors which includes all correct processors that did not yet receive and partially encrypt *m*. (See Section 3.3.2.)

A4.7. Every valid message *m* that is timely received by a correct processor is partially encrypted and relayed to all processors that are not in the path information of *m*. This implies that *m* is sent to all correct processors that did not yet receive and partially encrypt *m*. (See Section 3.3.2.)

A4.8. If the source is correct, in every destination processor *d*, a valid message from the source arrives in *d* within a real-time interval of length between τ_{min} and τ_{max} after the source started the PCFAP. ❑

Furthermore, we make the following additional assumptions for our PCFAPs:

A4.9. The component cryptographic keys f_j (with $1 \le j \le totalccks$) of function *F* are distributed among the processors in *N* by means of a cck-distribution that satisfies R4.1, R4.2, and R4.3. Furthermore, all correct processors know how the component cryptographic keys have been distributed.

A4.10. All correct processors in N agree upon the data that should be encrypted. Let $B(p_i)$ be the data that should be encrypted according to processor p_i, then for any pair of correctly functioning processors p_i and p_j, it holds that $B(p_i) = B(p_j)$.

A4.11. Any correctly functioning processor p_i of N is able to verify the correctness of every partially encrypted piece of data.

The above assumptions A4.9 through A4.11 can be satisfied as follows.

Since the user knows all the component cryptographic keys f_j (with $1 \leq j \leq totalccks$) of function F, the easiest way to satisfy assumption A4.9 is to let the user be responsible for the distribution of the component cryptographic keys of his/her secret cryptographic function F. Solving the key distribution problem is far from trivial, however, it falls beyond the scope of this chapter.

In order to satisfy assumption A4.10, before the start of the DCFAP, all correct processors in N should have reached agreement on the data, B, on which the secret cryptographic function F should be applied. How this agreement can be reached, has been discussed in Section 4.1.3.4.

Assumption A4.11 can be satisfied since any processor p_i is always able to retrieve the original data from the partially encrypted data, since for any component cryptographic function that has been applied to the original data, p_i knows the corresponding inverse component cryptographic function (since the corresponding component cryptographic key has been published, as has been stated in Section 4.2.2.). Since prior to the execution of the DCFAP, all correct processors have reached agreement about the data that should be encrypted, processor p_i is able to check the correctness of the partially encrypted data, by applying the inverse component cryptographic functions and comparing the result with the original data that has been agreed upon before the start of the DCFAP.

The lengths of the various communication phases of the PCFAP, as viewed from any correct processor, are given by A4.12. as follows.

A4.12. Every processor in the system may have a different view of time and of the start and end of communication phases in a PCFAP. For all k, with $1 \leq k \leq K$, L_k is the length of communication phase k, as viewed from i, where L_k is recursively defined by [18]

$$L_1 = (\tau_{max} - \tau_{min})\cdot(1+\rho)$$
$$L_2 = (\tau_{max} - \tau_{min})\cdot(1+\rho)^3 + (\Delta + \tau_{max} + \tau_{min})\cdot(1+\rho)$$
$$(\forall\, h \in [3,K]\colon L_h = [L_{h-1}\cdot(1+\rho)+\Delta]\cdot(1+\rho))$$

A PCFAP satisfying A4.12. can be designed such that it satisfies the conditions PCFAP1 and PCFAP2. However, usually, a PCFAP will be part of a DCFAP. The lengths of the communication phases of a DCFAP, viewed from a processor, are equal[19] or longer than the lengths of the communication phases of a PCFAP, viewed from that processor. This implies, that, if the PCFAP is used as a part of a DCFAP used to encrypt the hash value of a recovered re-created data fragment, the length of any communication phase in a PCFAP should be as long as that of the corresponding communication phase in the DCFAP. In other words, in that case, the length of the communication phases of the PCFAP should be as indicated by A4.13. (instead of A4.12.).

4.2.4.2. Description of a PCFAP based on a fault-tolerant multisignature scheme

In this section we will describe a PCFAP based on a fault-tolerant multisignature scheme.

The PCFAP is executed on a set, N, of N processors p_i ($1 \le i \le N$), up to T of which may behave maliciously, and consists of applying the secret cryptographic function F on the data fragment $B(p_x)$ held by the source p_x (for some x, with $1 \le x \le N$). For any j, with $1 \le j \le totalccks$, let f_j be a component cryptographic key of a secret cryptographic function F. Furthermore, let $f_j^{(-1)}$ be the corresponding component cryptographic key of the corresponding public cryptographic function $F^{(-1)}$. We will refer to any pair of corresponding component cryptographic keys $\{f_j, f_j^{(-1)}\}$ as a ***component cryptographic key pair***. With all component cryptographic keys f_j (respectively $f_j^{(-1)}$), with $1 \le j \le totalccks$, function F (respectively $F^{(-1)}$) can be generated. We assume that the component cryptographic keys of function F are kept secret, whereas the component cryptographic keys of key $F^{(-1)}$ are made public to all processors. We assume that A4.1 through A4.12 from Section 4.2.4.1. are satisfied.

For any processor $p_i \in N$, the set of all component cryptographic keys that it possesses is indicated by $cck(p_i)$. Thus:

D4.2. $(\forall\ p_i \in N :: cck(p_i) = \{f_j \mid f_j \textit{ is possessed by } p_i\})$

18. Notice that, in a PCFAP, K communication phases are required (for cck-distribution 2, $K = T^2+1$, whereas $T+1$ communication phases are sufficient for a BAP), since the correct processors in a PCFAP must satisfy the conditions PCFAP1 and PCFAP2, i.e., it is not sufficient for the correct processors only to decide on basis of equal sets of valid messages, instead, they should decide on basis of equal sets of valid messages that are fully (and not only partially) encrypted with secret cryptographic function F. Any valid message containing path information describing a path containing K processors is fully encrypted with function F. Provided such a message is timely received, and hence accepted in a correct processor, this correct processor will relay this message to all other processors in the system, such that these processors will timely receive and hence, accept this message, i.e. in or before the last communication phase (i.e., communication phase K). In this way, it is guaranteed that all correct processors accept the same set of fully encrypted messages. The decision taken in any correct processor is based only on accepted messages which contain path information describing a path containing $K+1$ processors.
19. Mostly, in a DCFAP, every communication phase L_i (in A4.13) will be **longer** than the same communication phase L_i (in A4.12) in a PCFAP. The length of a communication phase L_i ($1 \le i \le K$) is only **equal** in the following cases. Communication phase L_1 is equal for a PCFAP (A4.12) and a DCFAP (A4.13), only if $\tau_{max} = \tau_{min} = 0$ (This implies, that for all i, with $1 \le i \le K$, $L_i = 0$). For $2 \le i \le K$, L_i is also equal for a PCFAP and a DCFAP, if $\rho = 0$.

A PCFAP consists of a multicast process and a decision-making process. In general, during the multicast process of the PCFAP, every processor $a \in N$ receives a number of messages, say M_a ($M_a \geq 0$). Let *Messages* be the (infinite) set of all possible messages. For any processor $a \in N$, and $1 \leq i \leq M_a$, by $mess_i(a)$, we will denote the i-th message that processor a receives during the multicast process of the PCFAP. Here, $mess_i(a) \in$ *Messages*.

Such a message $mess_i(a)$ needs not always be valid. Faulty processors may corrupt a valid message, which may result in an invalid message. The decisions taken in a destination processor p_j ($1 \leq j \leq N$) about the result of encryption of the initial value $B(p_x)$ of the source p_x with secret cryptographic function F are based on the valid messages that p_j has accepted. We will now define what is meant by valid and invalid messages, respectively.

D4.3. Any message $mess_i(a) \in Messages$ (with $1 \leq i \leq M_a$) received by a certain processor $a \in N$ is a valid message iff $mess_i(a) \in ValidMessages$. Conversely, any message that is not valid is an invalid message.

Here, *ValidMessages* is the set of all possible valid messages, which will be defined below. In this definition, we will make use of the sets *Paths*, *MessageDigests*, *EncryptedPathParts*, and *ValidPathParts*. These sets have been defined in definitions D3.3, D3.9, D3.11, and D3.12 in Section 3.3.3.1, respectively. We will also use the set *ValidPaths*, which is given by

D4.4.
$$
\begin{aligned}
&(\forall\ (\underline{\boldsymbol{p}}) \in Paths :: validpath(\ (\underline{\boldsymbol{p}})\ ) \equiv \\
&\qquad ((length(\ (\underline{\boldsymbol{p}})\ ) \leq K \wedge (\forall\ a \in N :: occurrences(a, (\underline{\boldsymbol{p}})) \leq 1)) \\
&\quad \vee \qquad (length(\ (\underline{\boldsymbol{p}})\ ) = K{+}1 \wedge (\forall\ a \in N :: occurrences(a, prefix((\underline{\boldsymbol{p}}))) \leq 1)) \\
&\qquad) \\
&)
\end{aligned}
$$

i.e., function $validpath(\ (\underline{\boldsymbol{p}})\ ) \equiv$ TRUE if the length of path $(\underline{\boldsymbol{p}})$ is less than or equal to K, and every processor a occurs at most once in path $(\underline{\boldsymbol{p}})$, or if the length of path $(\underline{\boldsymbol{p}})$ is equal to K+1, and every processor a occurs at most once in the prefix of path $(\underline{\boldsymbol{p}})$. [20]

The set *EncryptedMessageValues* is identical to the set *MessageValues* in definition D3.2. (Notice that in a PCFAP, a message value may also be partially encrypted.) Furthermore, *SetOfComponentCryptKeys* is defined as the powerset of the sets of component cryptographic keys of secret cryptographic function *F*.

The set of all possible valid messages, *ValidMessages*, is defined by:

20. In a valid message with valid path information describing a path of length K+1, the identification of the last processor may occur twice in the path. This can be seen as follows.

Since the decisions of the correct processors about the result of encryption of the commonly known data is based on the fully encrypted message values of the messages that are accepted during the DCFAP, these (fully encrypted) messages are relayed to all processors, instead of only to the processors that did not yet receive the message. Hence, in a valid message, the identification of the last processor in the path may occur also in the prefix of the path.

D4.5. $ValidMessages = \{(emv;vpp) \mid emv \in EncryptedMessageValues \wedge vpp \in ValidPathParts \wedge messagedigest(vpp) = digest(retrievemv(emv,vpp))\ \}$

For any $emv \in EncryptedMessageValues$, and $vpp \in ValidPathParts$, the function $retrievemv(emv,vpp)$ is defined by

D4.6. $(\forall\ emv \in EncryptedMessageValues : \forall\ vpp \in ValidPathParts :$
$\forall\ (\boldsymbol{p}) \in Paths : \forall\ epp \in EncryptedPathParts ::$
$(vpp = ((\boldsymbol{p});epp) \Rightarrow$
$retrievemv(emv,vpp) = reversecrypt(emv, CCK((\boldsymbol{p}))))$
$)$

Here, for any $emv \in EncryptedMessageValues$, and $cckset \in SetOfComponentCryptKeys$, the function $reversecrypt(emv,cckset)$ is defined by

D4.7. $(\forall\ emv, emv' \in EncryptedMessageValues :$
$\forall\ cckset \in SetOfComponentCryptKeys ::$
$reversecrypt(emv,cckset) = emv')$
where
$emv' = F(f_{v(1)}{}^{(-1)}, F(f_{v(2)}{}^{(-1)}, F(\ldots, F(f_{v(m)}{}^{(-1)},emv)\ldots)))$
and
$cckset = \{f_{v(1)}, f_{v(2)}, \ldots, f_{v(m)}\}$

Furthermore, for any $(\boldsymbol{p}) \in Paths$, $CCK((\boldsymbol{p}))$ is defined as the set of component cryptographic keys possessed by the processors in path $(\boldsymbol{p})$, i.e.,

D4.8. $(\forall\ (\boldsymbol{p}) \in Paths ::(empty((\boldsymbol{p})) \Rightarrow CCK((\boldsymbol{p})) = \emptyset) \wedge$
$(\neg empty((\boldsymbol{p})) \Rightarrow (\exists(\boldsymbol{q}) \in Paths: \exists\ a \in N :: (\boldsymbol{p}) = (\boldsymbol{q};a) \wedge$
$CCK((\boldsymbol{p})) = CCK((\boldsymbol{q})) \cup cck(a))$
$)$

Here, the functions *empty* and *cck* have been defined in D3.4. and D4.2. respectively.

For any valid message, the functions *messvalue* and *path* are defined by

D4.9. $(\forall\ vm \in ValidMessages : \forall\ vpp \in ValidPathParts :$
$\forall\ emv \in EncryptedMessageValues::$
$(vm = (emv;vpp)) \Rightarrow$
$messvalue(vm) = retrievemv(emv,vpp) \wedge$
$path(vm) = pathfrompathpart(vpp)$
$)$

The function *pathfrompathpart* has been defined in D3.16 in Section 3.3.3.1.

Now, the following notation is introduced:

N4.3. Let $mess_i(a)$ be the i-th message that processor a receives $(1 \le i \le M_a)$. Assume that $mess_i(a) \in ValidMessages$, with $messvalue(mess_i(a)) = mv$, and $path(mess_i(a)) = (\boldsymbol{p})$. Such a valid message will be denoted as $m(mv,(\boldsymbol{p}))$.

Before we describe the PCFAP, we first discuss how messages are generated respectively how they are relayed.

Generating a message

The message generation process is performed by any correct source, say s ($s \in N$). Roughly, this message generation process consists of the following steps:

Gs1. The source s selects $B(s)$ as the message value $mv \in MessageValues$, the result of encryption of which it wants to communicate to all destination processors.

Gs2. For any processor $a \in N$ ($a \neq s$), which s wants to send a partially encrypted message to, s generates a valid message $vm \in ValidMessages$, where $vm = (cmv;vpp)$, with $cmv = crypt(mv,cck(s))$, and $vpp = ((s), K_{(s)}\{digest(mv) ; a\})$, and sends vm to a. Notice that, according to N4.3, $vm = m(mv, (s;a))$.

Here, for any $emv \in EncryptedMessageValues$, and $cckset \in SetOfComponentCryptKeys$, the function $crypt(emv,cckset)$ is defined by

D4.10. ($\forall\ emv, emv' \in EncryptedMessageValues$:
$\forall\ cckset \in SetOfComponentCryptKeys$ ::
$crypt(emv,cckset) = emv'$)
where
$emv' = F(f_{v(1)}, F(f_{v(2)}, F(\ldots, F(f_{v(m)},emv)\ldots)))$
and
$cckset = \{f_{v(1)}, f_{v(2)}, \ldots, f_{v(m)}\}$

Relaying a message

Roughly, the message relay process as it is performed by any correct processor $a \in N$ consists of the following steps:

Rs1. At receipt of a message, processor a checks if the received message is valid or not.

Rs2. If the received message is not valid, it is rejected by a.

Rs3. If the received message is valid, but received too late, or not received for the first time[21], it is also rejected by a.

Rs4. If the received message is valid, and it is timely received, and it is received for the first time, it is relayed by a in such a way that the resulting message is again a valid message. Any correct processor $a \in N$, that wants to relay a valid message $vm \in ValidMessages$ to a certain processor $b \in N$ (with $validpath(path(vm);b)$), sends $relay_{(a,b)}(vm)$ to b (where function *relay* is defined below).

Function *relay* mentioned in step Rs4 is formally defined as follows.

Assume that a correct processor $a \in N$ wants to relay a valid message $vm \in ValidMessages$ to a certain processor $b \in N$ (with $validpath(path(vm);b)$), then it performs the function $relay_{(a,b)}(vm)$ which is defined by

D4.11. ($\forall\ vm \in ValidMessages$: $\forall\ emv \in EncryptedMessageValues$: $\forall\ (\underline{\mathbf{p}}) \in Paths$:
$\forall\ epp \in EncryptedPathParts$::
$(vm = (emv, ((\underline{\mathbf{p}}); epp)) \wedge validpath(path(vm);b)) \Rightarrow$

21. More precise definitions of valid messages that are received too late, or not received for the first time are given in D4.14. and Section 3.3.6. and 3.3.7., respectively.

$$relay_{(a,b)}(vm) = (crypt(emv, cck(a) \setminus CCK((\underline{\mathbf{p}})));((\underline{\mathbf{p}};a);K_{(a)}\{epp;b\})))$$

Notice that the function *relay* can also be defined by

D4.12. $(\forall\ a \in N : \forall\ emv \in EncryptedMessageValues: \forall\ (\underline{\mathbf{p}}) \in Paths::$
$length((\underline{\mathbf{p}})) \geq 2 \Rightarrow relay_{(last((\underline{\mathbf{p}})),a)}(m(emv,(\underline{\mathbf{p}}))) =$
$m(crypt(emv, cck(last((\underline{\mathbf{p}})))\setminus CCK(prefix((\underline{\mathbf{p}})))),(\underline{\mathbf{p}};a))\ )$

We also use the following definitions:

D4.13. A valid message *m* is ***received in time*** in a correct processor *c* if it arrives in *c* in or before communication phase *i* ($1\leq i \leq K$) of *c*'s sub-PCFAP, and the path information of *m* describes a path containing at least *i*+1 processors.

D4.14. A valid message *m* is ***received too late*** in a correct processor *c* if it arrives in *c* after *c* has concluded communication phase *i* ($1 \leq i \leq K$) of its sub-PCFAP, and the path information of *m* describes a path containing at most *i*+1 processors.

D4.15. A valid message *m*, sent directly from the source to a destination processor *d* is ***sent in time***, if, after the source started the PCFAP (by sending the first valid message of the PCFAP), *m* arrives in *d* within a real-time interval of length between τ_{min} and τ_{max}. Similarly, a valid message *m*, relayed by a destination processor *j* in communication phase *i* of its sub-PCFAP to another destination processor *k* is ***sent in time*** if *m* arrives in *k* within a real-time interval of length between τ_{min} and τ_{max} after *j* started communication phase *i* of its sub-PCFAP.

Before we give the description of the PCFAP, we make the following remarks.

- Assumption A4.4 states that a processor may concurrently execute different PCFAPs. Without further specification, it is assumed that messages of different PCFAPs can be distinguished, such that every correct processor can determine when it receives the first valid message of a new PCFAP. This implies that, given the contents of any received valid message, any correct processor should be able to uniquely identify the PCFAP to which this message belongs.

  In fact, a processor may concurrently execute different DCFAPs, each of which consists of a number of concurrently executed PCFAPs. So, in order to identify the PCFAP to which a received message belongs, every message in every PCFAP should carry an identification which states to which PCFAP of which DCFAP the message belongs.

  To *identify the DCFAP* to which a received message belongs, every DCFAP can be given a unique identification, e.g., a sequence number. DCFAPs based on a fault-tolerant multisignature scheme consist of execution of a number of PCFAPs.

  Since all PCFAPs belonging to a certain DCFAP have a unique source, the source of a received valid message uniquely *identifies the PCFAP* (within the identified DCFAP) to which that message belongs. The source processor of a certain PCFAP is the first processor in the path described by the path information of any valid message belonging to that PCFAP. So, the path information of a valid message can be used to identify the PCFAP (within the identified DCFAP) to which that message belongs.

To avoid confusion, in the following description of a PCFAP, we will omit the process of identifying, for each received valid message, the PCFAP and the DCFAP to which that message belongs, since this would only unnecessarily complicate the understanding of how the protocol works.

- In order to be able to decide whether or not a valid message m was received in time, according to D4.13, a correct processor that receives m in communication phase j ($1 \leq j \leq K$) of its sub-PCFAP, checks if j is smaller than the number of processors in the path information of m. We assume, that every processor $i \in N$ stores the clock values (of its own processor clock) at which the communication phases of its sub-PCFAP end in an array $endcommphase_i$.

  Initially, the processors do not know when a PCFAP will start, and hence, any processor $i \in N$ does not know these clock values until i receives the first valid message from the PCFAP. Hence, for any processor $i \in N$, and any communication phase j, with $1 \leq j \leq K$, $endcommphase_i(j)$ is initialized to ∞.

  Any correct processor $i \in N$ will start the first communication phase of its sub-PCFAP as soon as processor i receives the first valid message from the PCFAP, say at clock value $C(i,t)$ [22] of its processor clock. Then, processor i is able to calculate at which clock values $C(i,t')$ of its processor clock each of the K communication phases of its sub-PCFAP ends, since, by A4.12., i knows a priori the length L_k ($1 \leq k \leq K$) of every communication phase of its sub-PCFAP. For all h, with $1 \leq h \leq K$, it holds that

$$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

  By A4.12., for all k, with $1 \leq k \leq K$, L_k is the length of communication phase k, as viewed from i, where L_k is recursively defined by:

$$L_1 = (\tau_{max} - \tau_{min}) \cdot (1+\rho)$$
$$L_2 = (\tau_{max} - \tau_{min}) \cdot (1+\rho)^3 + (\Delta + \tau_{max} + \tau_{min}) \cdot (1+\rho)$$
$$(\forall\, h \in [3,K]: L_h = [L_{h-1} \cdot (1+\rho) + \Delta] \cdot (1+\rho))$$

- It is assumed that one processor (i.e., the source s) is triggered to start the PCFAP. This may be due to some external event (e.g., the receipt of a message belonging to a DCFAP which processor s did not yet participate in) or because s's processor clock reaches a certain predefined clock value. We will not further specify this event in the protocol given below.

Now, a PCFAP based on a fault-tolerant multisignature scheme can be described as follows:
Initially, for each processor $i \in N$, for all j, with $1 \leq j \leq K$, $endcommphase_i(j) = \infty$

22. $C(i,t)$ is the value of processor i's clock at time t. $C(i,t)$ has been defined in assumption A4.2.

For some processor $s \in N$:

Event: *Processor s is triggered to start the PCFAP at time C(s,t)*

Action: *Processor s acts as the source and generates and sends a message m(emv,(s;k)) to every processor $k \in N \setminus \{s\}$. The source accepts a copy of its own message in order to use it in the decision-making process. Furthermore, the source calculates at which clock values C(s,t') of its processor clock each of the K communication phases of its sub-PCFAP ends. For all h, with $1 \le h \le K$, it holds that*

$$endcommphase_s(h) = C(s,t) + \sum_{k=1}^{h} L_k$$

For each processor $i \in N$:

Event: *$C(i,t) \ge endcommphase_i(K)$*

Action: *processor i decides on basis of the message values of the messages it has accepted, and which contain path information describing a path containing K + 1 processors. If one or more message values have been properly encrypted, then the decision taken in i consists of random selection of one of these message values, otherwise, processor i decides a default value.*

Event: *message $mess_j(i)$ (with $1 \le j \le M_i$) received at time C(i,t)*

Action:

1. *Check if $mess_j(i)$ is a valid message, i.e., check if $mess_j(i) \in$ ValidMessages*
   If invalid($mess_j(i)$) then reject $mess_j(i)$ and abort, i.e., do no perform further actions on $mess_j(i)$.
   If valid($mess_j(i)$), then $mess_j(i)$ = m(emv,($\underline{\mathbf{q}}$)) for some path ($\underline{\mathbf{q}}$) $\in$ ValidPaths and some emv $\in$ EncryptedMessageValues. If last(($\underline{\mathbf{q}}$)) $\neq$ i, then reject $mess_j(i)$ and abort, otherwise $mess_j(i)$ = m(emv,($\underline{\mathbf{p}}$;i)) for path ($\underline{\mathbf{p}}$) $\in$ ValidPaths with ($\underline{\mathbf{p}}$) = prefix(($\underline{\mathbf{q}}$)).
2. *Check if a valid message $mess_k(i)$ (with $1 \le k < j$) with path($mess_k(i)$) = ($\underline{\mathbf{p}}$;i) has already been accepted.*
   If such a message has already been accepted, then reject $mess_j(i)$ and abort.
3. *Check if m(emv,($\underline{\mathbf{p}}$;i)) has been received in time, i.e., check if C(i,t) $\le$ $endcommphase_i$(length(($\underline{\mathbf{p}}$;i))-1). If m(emv,($\underline{\mathbf{p}}$;i)) has not been received in time, i rejects m(emv,($\underline{\mathbf{p}}$;i)).*
4. *Check if the (partially encrypted) message value emv of m(emv,($\underline{\mathbf{p}}$;i)) has been properly encrypted, i.e., check if reversecrypt(emv, CCK(($\underline{\mathbf{p}}$;i))) = B(p_i). If emv has been properly encrypted, then i accepts m(emv,($\underline{\mathbf{p}}$;i)), otherwise i rejects m(emv,($\underline{\mathbf{p}}$;i)).*
5. *If m(emv,($\underline{\mathbf{p}}$;i)) is the first valid message that i accepts, i starts the first communication phase of its sub-PCFAP at receipt of m(emv,($\underline{\mathbf{p}}$;i)). Processor i now calculates at which clock values C(i,t') of its processor clock each of the K communication phases of its sub-PCFAP ends.*

For all h, with 1 ≤ h ≤ K, it holds that

$$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

6. *If i accepted m(emv,(**p**;i)) and path(m(emv,(**p**;i))) contains fewer than K processors, for every processor h that is not in path(m(emv,(**p**;i))), i signs message m(emv,(**p**;i)) and i relays m(emv,(**p**;i)) to h, immediately after m(emv,(**p**;i)) was received by i. As stated earlier in this section, relaying message m(emv,(**p**;i)) to any processor h results in m(crypt(emv,cck(i) \ CCK((**p**))),(**p**;i;h)) being sent to h.*
7. *If i accepted m(emv,(**p**;i)) and path(m(emv,(**p**;i))) contains K processors, for every processor h ∈ N, i signs message m(emv,(**p**;i)) and i relays m(emv,(**p**;i)) to h, immediately after m(mv,(**p**;i)) was received by i. Since the message value emv has been encrypted with all component cryptographic keys of F, cck(i) \ CCK((**p**)) = ∅, thus, crypt(emv,cck(i) \ CCK((**p**))) = emv, and hence, relaying message m(emv,(**p**;i)) to any processor h results in m(emv,(**p**;i;h)) being sent to h.*

All correct processors are assumed to execute the above-described protocol. The behaviour of faulty processors, however, may deviate arbitrarily from the above-described protocol.

Below, we only give an outline of the proof that the above protocol satisfies the conditions PCFAP1 and PCFAP2, since the proof is very similar to the one given in Section 3.3.9., which proves that authenticated self-synchronizing BAPs satisfy the interactive consistency conditions. However, the following two essential differences exist between a PCFAP and a BAP:

1. In contrast to a BAP, the decisions of the correct processors in a PCFAP are not based on the set of message values of *all* messages that are accepted during execution of the protocol, but only on the set of message values of the accepted valid messages with path information describing a path containing the identifications of K+1 processors.
2. In contrast to a BAP, in every communication phase of the PCFAP, messages are not only relayed, but also partially encrypted.

First, consider the case that the PCFAP is initiated by a correct source.

Notice that if the source of the PCFAP is correct, then the message value of every valid message with path information describing a path containing K processors is equal to the result of encryption of the initial value possessed by the source with all component cryptographic keys of F [23]. It is guaranteed that such a valid message is generated and received and accepted by at least one correct processor c, provided that at least one message from the source has travelled along a path of K-2 correct processors to processor c.

In the presence of at most T faulty processors, it is guaranteed that **at least one** correct processor c will accept such a message, because

a. a valid message from the source is timely received and accepted in at least one correct processor, since the source initiates the PCFAP by multicasting a message to all N-1 ≥ T+1 other processors in the system.

b. every valid message with path information describing a path of length smaller than K which is accepted by a correct processor is relayed to at least one correct processor. Viz., every valid message *m*(*emv*,(***p***)), for which the length of path (***p***) is smaller than K, is relayed to all processors, the identification of which is not present in path (***p***), hence, every valid message *m*(*emv*,(***p***)) is relayed to N - *length*((***p***)) processors. Since N - *length*((***p***)) > N - K ≥ T+1, and the number of faulty processors is less than or equal to T, every valid message is relayed to at least one correct processor.

By having every correct processor that has accepted a valid message with path information describing a path containing K processors (i.e., a valid message of which the message value is equal to the result of encryption of the initial value held by the source with function F) relay this message to all processors in the system (including itself !), it is guaranteed that all correct processors will accept at least one valid message of which the message value is equal to the result of encryption of the initial value held by the source.

The decision taken in any correct processor c is based on the set of message values of the valid messages with path information describing a path of length K+1, which c has accepted during execution of the multicast process of the PCFAP. Let $F(B)$ be the result of encryption of the initial value B held by the source with function F, then execution of the PCFAP guarantees that the set of message values on which any correct processor bases its decision always contains the value $F(B)$. However, the set of message values on which a correct processor bases its decision cannot contain other message values, since any message received by a correct processor which contains a message value that, after having been decrypted, differs from B will be rejected by that correct processor, since it fails to pass the test which determines whether the message has been properly encrypted.

Now consider the case that the PCFAP is initiated by a set of colluding faulty processors.

In this case, since the source is faulty, satisfaction of condition PCFAP2 is trivial, and hence, we only have to prove that the PCFAP satisfies condition PCFAP1. However, since we assumed that the correct processors have a priori reached agreement about the data, B, that should be encrypted, any message received by a correct processor which contains a message value that, after having been decrypted, differs from B will be

23. Notice that any set of at least K processors possesses all component cryptographic keys of the secret cryptographic function F (See also Section 4.2.5.). Since, in a PCFAP, the cryptographic algorithm F is applied on the message value of any valid message that contains path information describing a path (***p***) of K+1 processors, with all component cryptographic keys possessed by the K processors in *prefix*((***p***)), the message value of such a message is equal to the result of application of the secret cryptographic function F on the initial value held by the source.(Notice that the message value of any valid message containing path information describing a path of K processors is not encrypted, but such a message is simply signed and distributed to all other processors in the system. The decision taken in each correct processor is based on the set of accepted valid messages containing path information describing a path of K+1 processors).

rejected by that correct processor, since it fails to pass the test which determines whether the message has been properly encrypted.

So, any correct processor will only accept a received valid message, if it contains a message value that, after having been decrypted, is equal to B. If any correct processor accepts a received valid message during the PCFAP, then, analoguously to the above case (i.e., the case in which the PCFAP is initiated by a correct source), we can prove that all correct processors will decide $F(B)$. If no correct processor ever accepts a received message during the PCFAP, then all correct processors will decide the same default value. Hence, satisfaction of condition PCFAP1 is guaranteed.

From the above, we may conclude that the PCFAP always satisfies PCFAP1 and PCFAP2. A formal proof of this is left to the reader.

4.2.4.3. Description of a DCFAP based on a fault-tolerant multisignature scheme

In the previous section, we have given a description of a PCFAP that is part of a DCFAP used to sign a re-created recovered data fragment. The corresponding DCFAP will be presented in this section.

A DCFAP has similarities with the authenticated self-synchronizing ICA given in Section 3.4.1. In fact, a DCFAP runs on a system, N, of N processors, and consists of execution of N PCFAPs. Every processor $p \in N$ is the source in one of the PCFAPs belonging to the DCFAP. Of these PCFAPs, for any processor $p \in N$, we will denote the PCFAP in which processor p acts as the source as PCFAP$_p$.

For DCFAPs based on a fault-tolerant multisignature scheme, the following definitions are used:

D4.16. A valid message m is ***received in time*** in a correct processor c if it arrives in c in or before communication phase i ($1 \leq i \leq K$) of c's sub-DCFAP, and the path information of m describes a path containing at least i+1 processors.

D4.17. A valid message m is ***received too late*** in a correct processor c if it arrives in c after c has concluded communication phase i ($1 \leq i \leq K$) of its sub-DCFAP, and the path information of m describes a path containing at most i+1 processors.

D4.18. A valid message m, sent directly from the source to a destination processor d is ***sent in time***, if, after the source started the DCFAP (by sending the first valid message of the DCFAP), m arrives in d within a real-time interval of length between τ_{min} and τ_{max}. Similarly, a valid message m, relayed by a destination processor j in communication phase i of its sub-DCFAP to another destination processor k is ***sent in time*** if m arrives in k within a real-time interval of length between τ_{min} and τ_{max} after j started communication phase i of its sub-DCFAP.

Before we give the description of the DCFAP, we make the following remarks.

- In a DCFAP, the processors do not know a priori when the DCFAP starts. It is assumed that at least one processor, say p, is triggered to initiate the DCFAP (e.g., due to the occurrence of some external event, or to its processor clock reaching a certain predefined value). This processor p initiates the DCFAP by starting its own PCFAP$_p$ which is part of the DCFAP (i.e., processor p acts as the source and generates and sends a message $m(mv,(p;k))$ to every processor $k \in N \setminus \{i\}$). Simultaneously, processor p starts the first communication phase of its sub-PCFAP of any other PCFAP$_j$ ($j \in N \setminus \{p\}$).

- The other processors are unaware of the start of the DCFAP until they receive the first valid message of that DCFAP. At receipt of this message, say from PCFAP$_p$, any processor $i \in N \setminus \{p\}$ will participate in PCFAP$_p$ (by starting the first communication phase of its sub-PCFAP of PCFAP$_p$). Simultaneously, processor i will also start the first communication phase of its sub-PCFAP of any other PCFAP$_j$ ($j \in N \setminus \{p\}$). Processor i starts its own PCFAP$_i$ by generating and sending a message $m(mv,(i;k))$ to every processor $k \in N \setminus \{i\}$.

- A DCFAP based on a fault-tolerant multisignature scheme consists of execution of a number of PCFAPs on a set of processors. In its turn, each PCFAP consists of a set of sub-PCFAPs each running on a different processor. The set of all sub-PCFAPs of the PCFAPs belonging to a DCFAP and running on a certain processor p ($p \in N$) will be referred to as the ***sub-DCFAP of processor p***.

- Since every processor $p \in N$ starts all its sub-PCFAPs of the PCFAPs belonging to a DCFAP simultaneously, it is justified to speak of communication phase j ($1 \le j \le K$) of the sub-DCFAP of processor p.

- In order to be able to decide whether or not a valid message m of a certain PCFAP$_s$ belonging to the DCFAP was received in time, according to D4.16., a correct processor that receives m in communication phase j ($1 \le j \le K$) of its sub-DCFAP, checks if j is smaller than the number of processors in the path information of m. We assume, that every processor $i \in N$ stores the clock values at which the communication phases of its sub-DCFAP end in an array $endcommphase_i$.

  Initially, the processors do not know when a DCFAP will start, and hence, any processor $i \in N$ does not know these clock values until i receives the first valid message from the DCFAP. Hence, for any processor $i \in N$, and any communication phase j of i's sub-DCFAP, with $1 \le j \le K$, $endcommphase_i(j)$ is initialized to ∞.

  As soon as processor i receives the first valid message from the DCFAP, say at clock value $C(i,t)$ of its processor clock, i starts the first communication phase of its sub-DCFAP. Then, processor i is able to calculate at which clock values $C(i,t')$ of its processor clock each of the K communication phases of its sub-DCFAP ends, since, as shown below, i knows a priori the length L_k ($1 \le k \le K$) of every communication phase of its sub-DCFAP. For all h, with $1 \le h \le K$, it holds that

$$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

  By A4.13., for all k, with $1 \le k \le K$, L_k is the length of communication phase k of i's sub-DCFAP, as viewed from i, where L_k is recursively defined by:

$$L_1 = 2\tau_{max} \cdot(1+\rho)$$
$$L_2 = 2\tau_{max} \cdot(1+\rho)^3 + \Delta \cdot(1+\rho)$$
$$(\forall\, h \in [3,K]: L_h = [L_{h-1}\cdot(1+\rho)+\Delta]\cdot(1+\rho))$$

- If all PCFAPs that are part of the DCFAP should have terminated according to a certain processor $i \in N$, i decides on the outcome of the DCFAP on basis of the decisions calculated in the different PCFAPs of the DCFAP.

- It is assumed that processor i can distinguish between messages belonging to different DCFAPs. In the DCFAP described below, we will omit the identification of the DCFAP from the messages, since this unnecessarily complicates the description of the algorithm.

A DCFAP based on a fault-tolerant multisignature scheme can be described as follows:
Initially, for each processor i ∈ N, for all j, with 1 ≤ j ≤ K, endcommphase$_i$(j) = ∞

For some (possibly more than one) processor s ∈ N:
Event: *Processor s is triggered to start the DCFAP at time C(s,t).*
Action: *Processor s acts as the source in PCFAP$_s$ belonging to the DCFAP and generates and sends a message m(emv,(s;k)) to every processor k ∈ N \ {s}. The source accepts a copy of its own message in order to use it in the decision-making process of PCFAP$_s$.*
Simultaneously, the source starts the first communication phase of its sub-DCFAP. Furthermore, the source calculates at which clock values C(s,t') of its processor clock each of the K communication phases of its sub-DCFAP ends. For all h, with 1 ≤ h ≤ K, it holds that

$$endcommphase_s(h) = C(s,t) + \sum_{k=1}^{h} L_k$$

For each processor i ∈ N:
Event: *C(i,t) ≥ endcommphase$_i$(K)*
Action: *For every k ∈ N, processor i decides the outcome of PCFAP$_k$ on basis of the message values of the messages which i has accepted in PCFAP$_k$, and which contain path information describing a path containing K+1 processors. Then, processor i decides the outcome of the DCFAP on basis of the decisions from all of the PCFAP$_k$ (for any k ∈ N) it has calculated. The outcome of the DCFAP is decided as follows. All default decisions of the PCFAPs are rejected. If the remaining decisions of the PCFAPs are all equal, then one of them is taken as outcome of the DCFAP, otherwise, the outcome of the DCFAP is a default value.*

Event: *message mess$_j$(i) (with 1 ≤ j ≤ M$_i$) received at time C(i,t)*
Action: 1. *Check if mess$_j$(i) is a valid message, i.e., check if mess$_j$(i) ∈ ValidMessages.*
If invalid(mess$_j$(i)) then reject mess$_j$(i) and abort, i.e., do no perform further actions on mess$_j$(i).

*If valid(mess$_j$(i)), then mess$_j$(i) = m(emv,(**q**)) of some PCFAP$_s$ belonging to the DCFAP (with s = first((**q**)), for some path (**q**) ∈ ValidPaths and some emv ∈ EncryptedMessageValues. If last((**q**)) ≠ i, then reject mess$_j$(i) and abort, otherwise mess$_j$(i) = m(emv,(**p**;i)) for path (**p**) ∈ ValidPaths with (**p**) = prefix((**q**)).*

2. *Check if a valid message mess$_k$(i) (with 1 ≤ k < j) with path(mess$_k$(i)) = (**p**;i) has already been accepted.*
   If such a message has already been accepted, then reject mess$_j$(i) and abort.
3. *Check if m(emv,(**p**;i)) has been received in time, i.e., check if C(i,t) ≤ endcommphase$_i$(length((**p**;i))-1). If m(emv,(**p**;i)) has not been received in time, i rejects m(emv,(**p**;i)).*
4. *Check if the (partially encrypted) message value emv of m(emv,(**p**;i)) has been properly encrypted, i.e., check if reversecrypt(emv, CCK((**p**;i)) = B(p$_i$). If emv has been properly encrypted, then i accepts m(emv,(**p**;i)), otherwise, i rejects m(emv,(**p**;i)).*
5. *If m(emv,(**p**;i)) is the first valid message of the DCFAP that i accepts, i starts the first communication phase of its sub-DCFAP at receipt of m(emv,(**p**;i)). Processor i now calculates at which clock values C(i,t') of its processor clock each of the K communication phases of its sub-DCFAP ends. For all h, with 1 ≤ h ≤ K, it holds that*

$$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

   Furthermore, processor i is triggered to start its own PCFAP (PCFAP$_i$) belonging to the DCFAP. Hence, processor i acts as the source in PCFAP$_i$ belonging to the DCFAP and generates and sends a message m(emv,(i;k)) to every processor k ∈ N \ {i}. Processor i accepts a copy of its own message in order to use it in the decision-making process of PCFAP$_i$.
6. *If i accepted m(emv,(**p**;i)) and path(m(emv,(**p**;i))) contains fewer than K processors, for every processor h that is not in path(m(emv,(**p**;i))), i signs message m(emv,(**p**;i)) and i relays m(emv,(**p**;i)) to h, immediately after m(emv,(**p**;i)) was received by i. As stated in the previous section, relaying message m(emv,(**p**;i)) to any processor h results in m(crypt(emv,cck(i) \ CCK((**p**))),(**p**;i;h)) being sent to h.*
7. *If i accepted m(emv,(**p**;i)) and path(m(emv,(**p**;i))) contains K processors, for every processor h ∈ N, i signs message m(emv,(**p**;i)) and i relays m(emv,(**p**;i)) to h, immediately after m(emv,(**p**;i)) was received by i. Since the message value emv has been encrypted with all component cryptographic keys of F, cck(i) \ CCK((**p**)) = ∅, thus, crypt(emv,cck(i) \ CCK((**p**))) = emv, and hence, relaying message m(emv,(**p**;i)) to any processor h results in m(emv,(**p**;i;h)) being sent to h.*

All correct processors are assumed to execute the above-described protocol. The behaviour of faulty processors, however, may deviate arbitrarily from the described protocol.

Assume a DCFAP based on a fault-tolerant multisignature scheme is executed in a fully-connected synchronous system *N* consisting of *N* processors (including a source *s*), up to *T* of which may behave maliciously. We assume that for any PCFAP belonging to the DCFAP, assumptions A4.1 through A4.3, A4.5 through A4.11, and A4.13 and A4.14 are satisfied, where A4.13 and A4.14 are given by

A4.13. Every processor in the system may have a different view of time and of the start and end of communication phases in the DCFAP. For all *i*, with $1 \le i \le K$, for any correct processor $p \in N$, let L_i be the length of communication phase *i* of *p*'s sub-DCFAP, as viewed from *p*. Then:

$$L_1 = 2\tau_{max} \cdot (1+\rho)$$

$$L_2 = 2\tau_{max} \cdot (1+\rho)^3 + \Delta \cdot (1+\rho)$$

$$(\forall\, h \in [3,K]\colon L_h = [L_{h-1} \cdot (1+\rho) + \Delta] \cdot (1+\rho))$$

A4.14. A processor may concurrently execute different DCFAPs. We assume that messages of different DCFAPs can be distinguished, such that every correct processor can determine when it receives the first valid message of a new DCFAP.

Analogously as it is done in the previous section for PCFAPs, it can be proven that provided that, in a DCFAP, the above-mentioned assumptions hold, then the conditions DCFAP1 and DCFAP2 given in Section 4.1.3.4., are satisfied. The verification of this is left to the reader.

Notice that the DCFAP described above also satisfies Pr1 through Pr3 stated in Section 4.1.5. as follows:

Pr1. malicious behaviour of up to *T* processors from *N* does not inhibit application of cryptographic function *F*.

Pr2. any group of *N*-*T* or more processors is capable of applying function *F*.

Pr3. *T* or fewer colluding processors from *N* are unable to compute the secret cryptographic function *F*, or to apply *F* without the help of one or more correct processors.

This can be seen as follows.

We first show that Pr1 is satisfied. The DCFAP described above consists of execution of *N* PCFAPs, one per processor. Assume that at most *T* processors are faulty. Then, in every PCFAP that is initiated by a correct source, the decision taken in all correct processors equals the result of encryption of the initial value held by the source with function *F*. In every PCFAP, that is initiated by a set of colluding faulty processors, the decision taken in all correct processors is either equal to a default value or it equals the result of encryption of the initial value held by the source with function *F*. From the above description of the DCFAP, it is clear that, hence, malicious behaviour of up to *T* processors from *N* does not inhibit application of cryptographic function *F*. Hence, Pr1 is satisfied.

Pr2 is satisfied, since, as shown above, by means of execution of a DCFAP, a group of *N* processors, up to *T* of which may behave maliciously, is capable of applying function *F*.

That Pr3 is satisfied follows directly from property R4.3. of the applied component cryptographic key distribution.

4.2.5. Efficiency considerations on PCFAPs based on a fault-tolerant multisignature scheme

The PCFAP described in Section 4.2.4.2. is started at a source processor, and proceeds along several paths of processors, each of which encrypts the partially encrypted data fragment with the new component cryptographic keys it possesses. The protocol continues until the cryptographic algorithm F has been applied with all the *totalcck* component cryptographic keys of the cryptographic function F. For PCFAPs based on either of the cck-distributions proposed in Section 4.2.3., the maximum length *MaxL* of the paths of processors that perform part of the encryption during the PCFAP is equal to the number of communication phases, K, of the PCFAP. Hence, for cck-distribution 1, $MaxL = T+1$, whereas for cck-distribution 2, $MaxL = T^2+1$. In this section, it will be shown that, for cck-distribution 2, by carefully choosing the paths of processors followed by the PCFAP, PCFAPs based on cck-distribution 2 can be constructed for which $MaxL = T+1$. For $T > 1$, such PCFAPs with $MaxL = T+1$ require fewer communication phases than PCFAPs with $MaxL = T^2+1$, and hence, they are to be preferred to PCFAPs with $MaxL = T^2+1$.

In order to reduce *MaxL* for PCFAPs based on cck-distribution 2 from T^2+1 to $T+1$, we will look more closely how to minimize the length of the paths of processors followed by the PCFAP. We will define a ***completed path*** as a path containing a set of processors that together possess all component cryptographic keys of the secret cryptographic function. The PCFAP thus consists of a number of completed paths.

Any completed path starts with a source processor which possesses k different component cryptographic keys. In the worst case, every subsequent processor in the path has only one new component cryptographic key, and then, the length of the path (i.e., the number of processors in the completed path) is $T^2 + 1$. This is an upper bound on the length of a completed path.

A lower bound on the length of a completed path is $T + 1$, since, from Pr3 in Section 4.1.3.4., we know that at least $T + 1$ processors are required to encrypt data with any secret cryptographic function.

The PCFAP should be designed in such a way that, in every situation of up to T arbitrarily faulty processors, at least one completed path contains only correct processors.

The duration of the PCFAP depends on the length of the longest completed path followed by the protocol. In Section 4.2.5.1., we will show that it is possible to have the protocol follow only completed paths of length $T+1$, and still guarantee that the correct encryption is performed in the presence of up to T maliciously functioning processors. Provided that the PCFAP follows only completed paths of length $T+1$, the number of communication phases in the multicast process of the PCFAP can be reduced to $T+1$ (instead of the T^2+1 communication phases required in the original protocol). In this way, for $T > 1$, the overall execution time of the PCFAP is reduced.

Furthermore, the amount of data communication in the PCFAP depends on the number of completed paths followed by the protocol. In Section 4.2.5.2., we will describe a simple method by means of which the number of completed paths followed by the protocol remains small. However, as we will also show in Section 4.2.5.2., this method does not always yield the minimal number of completed paths.

4.2.5.1. Partial cryptographic function application protocols with completed paths of minimal length only

Since the duration of the PCFAP depends on the length of the longest completed path followed by the protocol, it is advantageous to minimize the length of the longest completed path. However, the PCFAP must always follow at least one completed path which contains correct processors only.

We will show that, provided the number of faulty processors does not exceed T, the set of all completed paths of length T+1 always contains a path with correct processors only.

In the rest of this section, we use the following lemma:

LEMMA 4.1.1.
Let S_1 , S_2, … S_{i+1} be disjoint sets. Let $\mathbf{S}$ be $\{S_1, \ldots, S_{i+1}\}$. Let M be the union of S_1 through S_{i+1}, i.e., $M = S_1 \cup S_2 \cup \ldots \cup S_{i+1}$. Let X be a subset of M containing at most i elements. Then:

$$(\exists\, S_j \in \mathbf{S} :: j \in [1,i+1] \wedge S_j \cap X = \varnothing) \qquad ❑$$

The proof of this lemma is straightforward. The lemma will be used to prove the following theorem:

THEOREM 4.1.
Assume we have a system consisting of a set N of processors, up to T of which may behave maliciously. For any i, with $1 \le i \le N$ (with $N = T \cdot (T+1)+1$) , let p_i be a processor in set N. For any j, with $1 \le j \le T\cdot (T+1)+1$, let f_j be a component cryptographic key of a secret cryptographic function F. With all component cryptographic keys f_j, with $1 \le j \le T\cdot (T+1)+1$, function F can be applied. We assume that A4.1 through A4.11 from Section 4.2.4.1. are satisfied. We assume that our PCFAP is performed by a group of N processors and consists of applying a secret cryptographic function F on data fragment B, by following only completed paths of minimal length T+1.

Then, at least one completed path of length T+1 contains correct processors only. ❑

For simplicity, we first prove Theorem 4.1 for the case T=2. After that, we generalize the proof for arbitrary T.

Proof for T=2:
For the case T=2, set $N = \{p_1, p_2, p_3, p_4, p_5, p_6, p_7\}$. The component cryptographic key

distribution is given in Table 4.2. The minimal length of a completed path is T+1 = 3.

Table 4.14: Proposed component cryptographic key distribution for $T = 2$

processor	possessed component cryptographic keys
p_1	f_1, f_7, f_6
p_2	f_2, f_1, f_7
p_3	f_3, f_2, f_1
p_4	f_4, f_3, f_2
p_5	f_5, f_4, f_3
p_6	f_6, f_5, f_4
p_7	f_7, f_6, f_5

Now, we can distinguish three different disjoint sets S_1 , S_2, and S_3 of T(=2) processors, each of which contains $T\cdot$ (T+1) (=6) component cryptographic keys. The sets and the corresponding component cryptographic keys possessed by them are given in Table 4.3.

Table 4.15: Disjoint sets of T processors, each with k-1 component cryptographic keys for the case T =2

set	processors in set	possessed component cryptographic keys
S_1	p_1, p_4	$f_1, f_2, f_3, f_4, f_6, f_7$
S_2	p_2, p_5	$f_1, f_2, f_3, f_4, f_5, f_7$
S_3	p_3, p_6	$f_1, f_2, f_3, f_4, f_5, f_6$

For each of the sets S_i (with $1 \le i \le 3$), the component cryptographic key not possessed by one of the processors in the set S_i, is possessed by each of the processors in the corresponding set U_i, where:

$$U_1 = \{p_5, p_6, p_7\}$$
$$U_2 = \{p_6, p_7, p_1\}$$
$$U_3 = \{p_7, p_1, p_2\}$$

E.g., component cryptographic key f_5, which is not possessed by one of the processors in S_1, is possessed by p_5, p_6, and p_7.

Since T=2, each of the sets U_1 through U_3 contains at least one correct processor.

Now, we define X to be a subset of N, containing all faulty processors of N. Since we assumed that the number of faulty processors does not exceed T, set X contains at most T(=2) processors. Then, by Lemma 4.1.1., at least one of the sets S_1 through S_3 has an empty intersection with set X, i.e., at least one of the sets S_1 through S_3 contains correct processors only.

So there is always a set S_j (with $1 \leq j \leq 3$) which does contain correct processors only, and its corresponding set U_j always contains a correct processor, say p_h (with $1 \leq h \leq 7$).

Now, any path following the processors in S_j and processor p_h is a completed path which has minimal length and contains correct processors only. ❑

Proof for arbitrary *T*:
Now we will prove Theorem 4.1 for arbitrary T. Set $N = \{p_i \mid 1 \leq i \leq T \cdot (T+1)+1\}$. The component cryptographic key distribution for arbitrary nonnegative T is given in Section 4.2.3.2. The minimal length of a completed path is T+1. Now, we can distinguish T+1 different disjoint sets S_i of T processors, each of which contains $T \cdot (T+1)$ component cryptographic keys. Set S_i (with $1 \leq i \leq T+1$) is defined as follows:

$$S_i = \{p_j \mid j = i + h \cdot (T+1) \wedge 0 \leq h \leq T-1\}$$

Set S_i (with $1 \leq i \leq T+1$) contains all component cryptographic keys of function F, except the fragment f_{i+T^2}. This fragment is possessed by each of the T+1 processors in the corresponding set U_i, where, for $1 \leq i \leq T+1$, U_i is defined by:

$$U_i = \{ p_j \mid j = (i + T^2 + h) \textbf{ mod } N \wedge 0 \leq h \leq T \}$$

Each of the sets U_i (with $1 \leq i \leq T+1$) contains at least one correct processor.

Now, we define X to be a subset of N, containing all faulty processors of N. Since we assumed that the number of faulty processors does not exceed T, set X contains at most T processors. Then, by Lemma 4.1.1., at least one of the sets S_1 through S_{T+1} has an empty intersection with set X, i.e., at least one of the sets S_1 through S_{T+1} contains correct processors only.

So there is always a set S_j (with $1 \leq j \leq T+1$) which does contain correct processors only, and its corresponding set U_j always contains a correct processor, say p_h (with $1 \leq h \leq T \cdot (T+1)+1$).

Now, any path following the processors in S_j and processor p_h is a completed path which has minimal length and contains correct processors only. ❑

4.2.5.2. A method to reduce the number of completed paths in partial cryptographic function application protocols

In Theorem 4.1, we assumed that the PCFAP may follow all completed paths of minimal length. In the protocol, there are $(T+1)^2$ different sets of T+1 processors. The processors in any of these sets can be followed in any order by the protocol, resulting in a large number of completed paths with minimal length.

In this section, therefore, we investigate how the number of completed paths followed by the PCFAP can be reduced. This reduces the amount of data communication needed by the protocol. We present a simple method by means of which the number of paths followed by the protocol remains small.

First of all, provided that all processors in a certain path are functioning correctly, since we assumed that the encryption function is commutative, the order in which the processors in a path are followed by the protocol, has no influence on the encryption result. Therefore, we reduce the number of paths followed by the protocol, by *requiring that every path followed by the protocol contains a unique set of processors (i.e., different from the sets of processors of other paths)*. This can easily be established e.g., by visiting the processors in every path in a fixed order.

However, the number of paths followed by the protocol can be reduced much further, as long as it is guaranteed that at least one path contains correct processors only (in the presence of up to T faulty processors).

We will use the following lemma:

LEMMA 4.2.1.
Let S_1, S_2, … S_{i+1} be non-empty sets, which have exactly one element x in common, i.e.,

$$(\exists_1\, x\colon (\forall\, j,k \in [1,i+1]{::}\ (j \neq k \Rightarrow S_j \cap S_k = x)))$$

Let $M \supseteq S_1 \cup S_2 \cup \ldots \cup S_{i+1}$, and $\mathbf{S}$ be $\{S_1, \ldots, S_{i+1}\}$. Let Y be a subset of $M \setminus \{x\}$ containing at most i elements. Then:

$$(\exists\, S_j \in \mathbf{S} :: j \in [1,i+1] \wedge S_j \cap Y = \varnothing) \qquad ❑$$

The proof of Lemma 4.2.1. is straightforward and follows directly from Lemma 4.1.1.

We will now describe our method of reducing the number of completed paths followed by our PCFAPs. We introduce the method for the case T=2, and thereafter generalize it for arbitrary T.

Assume we have a system consisting of a set N of processors, up to T of which may behave maliciously. For any i, with $1 \le i \le N$ (with $N = T\cdot(T+1)+1$) , let p_i be a processor in set N. For any j, with $1 \le j \le T\cdot(T+1)+1$, let f_j be a component cryptographic key of cryptographic function F. With all component cryptographic keys f_j, with $1 \le j \le T\cdot(T+1)+1$, function F can be applied. We assume that A4.1 through A4.11 from Section 4.2.4.1. are satisfied. We assume that our PCFAP is performed by a group of N processors and consists of applying a cryptographic function F on data fragment B, by following only completed paths of minimal length T+1.

The reduction method for the case *T*=2
For T=2, all completed paths of minimal length, each one containing a unique set of processors, are given in Table 4.4. We will show that with a selection of 6 completed paths from the set above, provided that N does not contain more than 2 faulty processors, there is always at least one path with correct processors only.

For any combination of up to 2 faulty processors in N, the PCFAP must follow at least one path containing correct processors only.

Initially, the paths P_3, P_5, and P_7 are included in the protocol. For any i with $1 \le i \le 9$, we define set S_i as the set containing the processors of path P_i. Sets S_3, S_5, and S_7 have only one processor (viz. p_7) in common. Let Y be the set of all faulty processors in $N \setminus \{p_7\}$. Since we assumed that T=2, Y does not contain more than two processors. By Lemma 4.2.1., at least one of the sets S_3, S_5 , and S_7 has an empty intersection with Y. So, provided that processor p_7 functions correctly, at least one of these three paths contains correct processors only.

We have to include other paths from Table 4.4 in the PCFAP in order to guarantee a path with only correct processors in the case that p_7 is faulty. Notice that the protocol already contains a path with only correct processors, if processor p_7 functions correctly. The paths we are going to include next need only be followed in the case that p_7 is faulty !

Table 4.16: Description of all different completed paths of minimal length for T=2

Path	Description
P_1	$p_1 \rightarrow p_4 \rightarrow p_5$
P_2	$p_1 \rightarrow p_4 \rightarrow p_6$
P_3	$p_1 \rightarrow p_4 \rightarrow p_7$
P_4	$p_2 \rightarrow p_5 \rightarrow p_6$
P_5	$p_2 \rightarrow p_5 \rightarrow p_7$
P_6	$p_2 \rightarrow p_5 \rightarrow p_1$
P_7	$p_3 \rightarrow p_6 \rightarrow p_7$
P_8	$p_3 \rightarrow p_6 \rightarrow p_1$
P_9	$p_3 \rightarrow p_6 \rightarrow p_2$

For this purpose, we select the paths P_2 and P_4. We assume that p_7 is faulty. Sets S_2 and S_4 (i.e., the sets containing the processors in path P_2 respectively P_4) have only 1 processor in common (viz. p_6). Let Y be the set of all faulty processors in $N \setminus \{p_6, p_7\}$. Since we assumed that T=2 and p_7 is faulty, Y does not contain more than 1 processor. By Lemma 4.2.1., at least one of the sets S_2 and S_4 has an empty intersection with Y. So, provided that processor p_6 functions correctly (and that processor p_7 is faulty), at least one of these paths contains correct processors only.

Now we have to include a final path from Table 4.4 in the protocol in order to guarantee the presence of a path with only correct processors in the case that both p_6 and p_7 are faulty. For this purpose, path P_1 is selected.

Thus the final protocol for $T = 2$ contains the paths P_1, P_2, P_3, P_4, P_5, and P_7.

It is worthwhile noticing that the described reduction method does **not** yield an optimal solution (i.e., a solution with a minimal number of completed paths of minimal length). For T=2 an optimal solution only requires 5 paths, e.g., a solution with paths P_1, P_3, P_5, P_7 and P_9 is also possible. The verification is left to the reader. It is, however, not straightforward to arrive at this solution. In fact, this problem can be described as a so-called ***covering problem*** (see, e.g., [Bree89, Wake90]), which is known to be NP-hard.

The reduction method for arbitrary *T*

For arbitrary T, all $(T + 1)^2$ completed paths P_i (for $1 \le i \le (T + 1)^2$) of minimal length, each one containing a unique set of processors, are defined as follows:

$$\forall\, j,k \in [1,T+1]::$$
$$P_{(j-1)\cdot(T+1)+k} \equiv p_j \rightarrow p_{j+(T+1)} \rightarrow p_{j+2\cdot(T+1)} \rightarrow \dots \rightarrow p_{j+(T-1)\cdot(T+1)} \rightarrow p_{j+(T-1)\cdot(T+1)+k}$$

The selection of the paths for arbitrary T is done analogously to the selection of paths for the case T=2. The selection of the paths is done in T+1 selection steps. In selection step i (for $1 \le i \le T+1$), T+2-i paths are selected.

In the first selection step, we select all T+1 paths in the set:

$$\{ P_{(j-1)\cdot(T+1)+k} \mid j+k = T+2 \wedge 1 \le j \le T+1 \}$$

By Lemma 4.2.1., we can prove that, provided that processor $p_{T\cdot(T+1)+1}$ functions correctly, at least one of these T+1 paths contains correct processors only.

In selection step i (with $1 \le i \le T+1$), we select all T+2-i paths in the set:

$$\{ P_{(j-1)\cdot(T+1)+k} \mid j+k = T+3-i \wedge 1 \le j \le T+1 \wedge 1 \le k \le T+1 \}$$

The final protocol for arbitrary T contains the paths in the set:

$$\{ P_{(j-1)\cdot(T+1)+k} \mid j+k = T+3-i \wedge 1 \le i \le T+1 \wedge 1 \le j \le T+1 \wedge 1 \le k \le T+1 \}$$

The number of paths followed by the protocol is equal to

$$\sum_{i=1}^{T+1} i$$

which is equal to $(T^2 + 3\cdot T + 2) / 2$.

4.2.6. Summary

In this section, we have described distributed cryptographic function application protocols based on a fault-tolerant version of the multisignature scheme in [Okam88]. These protocols can be used in order to make a dependable distributed data storage system in which data is stored in an encrypted form resilient to up to T arbitrarily faulty processors in the system.

The DCFAPs run on a system consisting of a number of processors, each of which possesses a number of component cryptographic keys. Data fragments are encrypted by relaying them along several completed paths of processors, each of which performs the encryption function on the partially encrypted data fragment with all the new component cryptographic keys it possesses. The DCFAP consists of execution of a number of PCFAPs. The PCFAP described in Section 4.2.4.2. consists of a multicast process with K communication phases followed by a decision-making process. The PCFAP is based on an authenticated self-synchronizing BAP. The difference between the PCFAP and the BAP is, that, while in the multicast process of a BAP, messages are simply exchanged between processors in order to have all correct processors reach agreement about the initial value held by the source, in the multicast process of the PCFAP, messages are encrypted while being exchanged in order to have all correct processors in the system reach agreement about the result of encryption of the initial value held by the source with the secret cryptographic function F.

We have considered several optimizations of the PCFAP. We have minimized the length of the completed paths that are followed by the PCFAPs. In this way, the required number of communication phases of the multicast process of the PCFAP is reduced to T+1, and thus, for $T > 1$, the overall execution time of the protocol is reduced. Furthermore, we have presented a simple method in order to reduce the number of completed paths of minimal length that are followed by the PCFAPs.

4.3. Distributed cryptographic function application protocols based on function sharing

In this section, we will describe distributed cryptographic function application protocols which are based on function sharing (discussed in Section 4.1.4.3.).

A DCFAP based on function sharing is defined on a set, $\mathcal{N}$, of N processors p_i ($1 \le i \le N$), up to T of which may behave maliciously. We assume that $N = 2T+1$. Furthermore, it is assumed[24] that all correct processors p_i in $\mathcal{N}$ have a priori reached agreement about the data, B, on which the secret cryptographic function F should be applied. We assume that the inverse cryptographic function $F^{(-1)}$ is publicly known[25]. For every processor p_i, let $B(p_i)$ be the data that should be encrypted according to p_i.

We assume that, by means of function sharing, the secret cryptographic function F has been split into N shadow functions F_i ($1 \le i \le N$), that have been distributed[26] among the N processors p_i. Each processor p_i is given one shadow function F_i.

Any DCFAP based on function sharing consists of execution of N authenticated[27] self-

24. This assumption is made for any DCFAP. See also Section 4.1.3.4.
25. Notice that this assumption requires F and $F^{(-1)}$ to be functions of an asymmetric cryptosystem.
26. The problem of distribution of the shadow functions is similar to the key distribution problem mentioned in Section 4.2.4.1. Both problems require secure distribution of confidential information. Although solving the problem of distribution of the shadow functions is certainly not trivial, it is beyond the scope of this chapter.
27. For arbitrary $N \ge 2T + 1$, an authenticated BAP is really needed, because authenticated BAPs tolerate an arbitrary number of faulty processors (thus $N \ge 2T+1$ suffices), in contrast to non-authenticated BAPs, which require $N \ge 3T+1$.

synchronizing Byzantine Agreement protocols BAP$_i$ ($1 \leq i \leq N$)[28]. Processor p_i acts as the source in BAP$_i$ in order to have $F_i(B(p_i))$ (i.e., the result of encryption of $B(p_i)$ with shadow function F_i) be communicated to all other processors in N. Since, for any i (with $1 \leq i \leq N$), BAP$_i$ is part of a DCFAP, the lengths of the communication phases of BAP$_i$ must be as long as the lengths of the communication phases of the DCFAP it is part of. The lengths of the communication phases of the BAP are given by A4.15.

After execution of all BAPs, every processor p_i acts as a combiner, i.e., p_i combines the partially encrypted results from the BAPs[29] in order to calculate the result of application of the secret cryptographic function F on the commonly available input data B. This result can only be obtained if:

1. Prior to the start of the DCFAP, all correct processors in N have reached agreement about data B. How this agreement can be reached, has been discussed in Section 4.1.3.4.
2. After the multicast processes of the BAPs that together make up the DCFAP have terminated, processor p_i possesses at least T+1 out of N correct evaluations of different shadow functions F_j of the commonly available input data B, otherwise, obtaining this result is computationally infeasible. We assume that if any processor $p_j \in N$ is functioning correctly, then p_j correctly communicates the evaluation of its shadow function F_j of the commonly available input data B to all other processors in N. So the system should contain at least T+1 correctly functioning processors. This implies, that, in the presence of up to T maliciously functioning processors, the system should contain at least 2T+1 processors. This requirement has already been satisfied, since we assumed that $N = 2T+1$.
3. After having combined T+1 shares into a result, any processor $p_i \in N$ has a means to verify whether this obtained result is correct or not. This requirement is easily satisfied, since we assumed every processor $p \in N$ to have the disposal over the original data B, that should be encrypted, and the publicly known inverse cryptographic function $F^{(-1)}$. Processor p_i will decide some obtained result X to be correct, iff $F^{(-1)}(X) = B$.

Notice that the above requirements 2 and 3 enable any processor $p_i \in N$ to obtain the result of application of secret cryptographic function F on data B. Requirement 2 guarantees that at least one of the combinations that p_i can generate yields a correct result, whereas requirement 3 enables processor p_i to select one of the correct combinations. Agreement between the correct processors about the result can be obtained by guaranteeing that all correct processors receive the same set of shares (by having them communicate their shares by means of an interactive consistency algorithm), having them

28. For any processor $p_i \in N$, we will denote the BAP in which processor p_i acts as the source as BAP$_i$.
29. For any $j \in [1,N]$, if the decision taken in a correct processor p_i after execution of BAP$_j$ (obtained by processor p_i) is correct, it will equal to $F_j(B(p_j))$. If processor p_i and p_j are both correct and have reached agreement about the data, B, that should be encrypted (respectively decrypted), then it holds that $B(p_i) = B(p_j) = B$, and hence, the decision taken in a correct processor after execution of BAP$_j$ is equal to $F_j(B)$.

check the correctness of the different possible combinations in the same, a priori agreed order, and having them decide the first combination that is evaluated to be correct, or, if such a combination cannot be found, having them decide a default value.

For DCFAPs based on function sharing, the following definitions are used:

D4.19. A valid message m is ***received in time*** in a correct processor c if it arrives in c in or before communication phase i ($1 \leq i \leq T+1$) of c's sub-DCFAP, and the path information of m describes a path containing at least $i+1$ processors.

D4.20. A valid message m is ***received too late*** in a correct processor c if it arrives in c after c has concluded communication phase i ($1 \leq i \leq T+1$) of its sub-DCFAP, and the path information of m describes a path containing at most $i+1$ processors.

D4.21. A valid message m, sent directly from the source to a destination processor d is ***sent in time***, if, after the source started the DCFAP (by sending the first valid message of the DCFAP), m arrives in d within a real-time interval of length between τ_{min} and τ_{max}. Similarly, a valid message m, relayed by a destination processor j in communication phase i of its sub-DCFAP to another destination processor k is ***sent in time*** if m arrives in k within a real-time interval of length between τ_{min} and τ_{max} after j started communication phase i of its sub-DCFAP.

Before we give the description of the DCFAP, we make the following remarks.

- In the DCFAP, the processors do not know a priori when the DCFAP starts. It is assumed that at least one processor, say p, is triggered to initiate the DCFAP (e.g., due to the occurrence of some external event, or to its processor clock reaching a certain predefined value). This processor p initiates the DCFAP by starting its own BAP$_p$ which is part of the DCFAP (i.e., processor p acts as the source and generates and sends a message $m(mv_p,(p;k))$ to every processor $k \in N \setminus \{i\}$). Here, $mv_p = F_p(B(p))$. Simultaneously, processor p starts the first communication phase of its sub-BAP of any other BAP$_j$ ($j \in N \setminus \{p\}$).

- The other processors are unaware of the start of the DCFAP until they receive the first valid message of that DCFAP. At receipt of this message, say from BAP$_p$, any processor $i \in N \setminus \{p\}$ will participate in BAP$_p$ (by starting the first communication phase of its sub-BAP of BAP$_p$). Simultaneously, processor i will also start the first communication phase of its sub-BAP of any other BAP$_j$ ($j \in N \setminus \{p\}$). Processor i starts its own BAP$_i$ by generating and sending a message $m(mv_i,(i;k))$ to every processor $k \in N \setminus \{i\}$, with $mv_i = F_i(B(i))$.

- A DCFAP based on function sharing consists of execution of a number of BAPs on a set of processors. In its turn, each BAP consists of a set of sub-BAPs each running on a different processor. The set of all sub-BAPs of the BAPs belonging to a DCFAP and running on a certain processor p ($p \in N$) will be referred to as the ***sub-DCFAP of processor p***.

- Since every processor $p \in N$ starts all its sub-BAPs of the BAPs belonging to a

DCFAP simultaneously, it is justified to speak of communication phase j ($1 \leq j \leq T+1$) of the sub-DCFAP of processor p.

- In order to be able to decide whether or not a valid message m of a certain BAP_s belonging to the DCFAP was received in time, according to D4.19., a correct processor that receives m in communication phase j ($1 \leq j \leq T+1$) of its sub-DCFAP, checks if j is smaller than the number of processors in the path information of m. We assume, that every processor $i \in N$ stores the clock values at which the communication phases of its sub-DCFAP end in an array $endcommphase_i$.

  Initially, the processors do not know when a DCFAP will start, and hence, any processor $i \in N$ does not know these clock values until i receives the first valid message from the DCFAP. Hence, for any processor $i \in N$, and any communication phase j of i's sub-DCFAP, with $1 \leq j \leq T+1$, $endcommphase_i(j)$ is initialized to ∞.

  As soon as processor i receives the first valid message from the DCFAP, say at clock value $C(i,t)$ of its processor clock, i starts the first communication phase of its sub-DCFAP. Then, processor i is able to calculate at which clock values $C(i,t')$ of its processor clock each of the $T+1$ communication phases of its sub-DCFAP ends, since, as shown below, i knows a priori the length L_k ($1 \leq k \leq T+1$) of every communication phase of its sub-DCFAP. For all h, with $1 \leq h \leq T+1$, it holds that

$$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

  By A4.15., for all k, with $1 \leq k \leq T+1$, L_k is the length of communication phase k of i's sub-DCFAP, as viewed from i, where L_k is recursively defined by:

$$L_1 = 2\tau_{max} \cdot (1+\rho)$$
$$L_2 = 2\tau_{max} \cdot (1+\rho)^3 + \Delta \cdot (1+\rho)$$
$$(\forall\, h \in [3,T+1] :: L_h = [L_{h-1} \cdot (1+\rho) + \Delta] \cdot (1+\rho))$$

- If all BAPs that are part of the DCFAP should have terminated according to a certain processor $i \in N$, i decides on the outcome of the DCFAP on basis of the decisions calculated in the different BAPs of the DCFAP. Processor i decides by acting as a combiner to combine the decisions taken in the different BAPs. The result of combination is equal to $F(B)$, provided that at least $T+1$ decisions equal the result of evaluation of a shadow function on the commonly available input data B.

- It is assumed that processor i can distinguish between messages belonging to different DCFAPs. In the DCFAP described below, we will omit the identification of the DCFAP from the messages, since this unnecessarily complicates the description of the protocol.

Then, a DCFAP based on function sharing can be described as follows:
Initially, for each processor $i \in N$, for all j, with $1 \leq j \leq T+1$, $endcommphase_i(j) = \infty$

For some (possibly more than one) processor $s \in N$:

Event: *Processor s is triggered to start the DCFAP at time C(s,t).*

Action: *Processor s acts as the source in BAP_s belonging to the DCFAP and generates and sends a message $m(mv_s,(s;k))$ to every processor $k \in N \setminus \{s\}$. Here, $mv_s = F_s(B(s))$. The source accepts a copy of its own message in order to use it in the decision-making process of BAP_s.*

Simultaneously, the source starts the first communication phase of its sub-DCFAP. Furthermore, the source calculates at which clock values C(s,t') of its processor clock each of the T+1 communication phases of its sub-DCFAP ends. For all h, with $1 \le h \le T+1$, it holds that

$$endcommphase_s(h) = C(s,t) + \sum_{k=1}^{h} L_k$$

For each processor $i \in N$:

Event: $C(i,t) \ge endcommphase_i(T+1)$

Action: *For every $k \in N$, processor i decides the outcome of BAP_k on basis of the message values of the messages which i has accepted in BAP_k. Then, processor i decides the outcome of the DCFAP on basis of the decisions from all of the BAP_k (for any $k \in N$) it has calculated. Processor i acts as a combiner in order to calculate F(B), i.e., the result of application of secret cryptographic function F on the commonly available input data B. Provided that at least T+1 decisions of the BAP_k ($k \in N$) equal the result of evaluation of shadow function F_k on the commonly available input data B, it is easy to calculate F(B).*

The decision taken in processor i is determined as follows. Processor i evaluates the correctness of any combination of T+1 shares in a fixed, a priori agreed order, until the first correct combination is encountered (in this case, this correct combination is taken as i's decision), or until no combination is left anymore (in this case, processor i decides a default value). Any combination X is decided to be correct iff $F^{(-1)}(X) = B$.

Event: *message $mess_j(i)$ (with $1 \le j \le M_i$) received at time C(i,t)*

Action:

1. *Check if $mess_j(i)$ is a valid message, i.e., check if $mess_j(i) \in$ ValidMessages.*

   If invalid($mess_j(i)$) then reject $mess_j(i)$ and abort, i.e., do no perform further actions on $mess_j(i)$.

   If valid($mess_j(i)$), then $mess_j(i) = m(mv,(\underline{\mathbf{q}}))$ of some BAP_s belonging to the DCFAP (with s = first(($\underline{\mathbf{q}}$)), for some path ($\underline{\mathbf{q}}$) $\in$ ValidPaths and some $mv \in$ MessageValues. If last(($\underline{\mathbf{q}}$)) $\neq$ i, then reject $mess_j(i)$ and abort, otherwise, $mess_j(i) = m(mv,(\underline{\mathbf{p}};i))$ for path ($\underline{\mathbf{p}}$) $\in$ ValidPaths with ($\underline{\mathbf{p}}$) = prefix(($\underline{\mathbf{q}}$)).

2. *Check if a valid message $mess_k(i)$ (with $1 \le k < j$) with path($mess_k(i)$) = ($\underline{\mathbf{p}}$;i) has already been accepted.*

   If such a message has already been accepted, then reject $mess_j(i)$ and abort.

3. *Check if m(mv,(**p**;i)) has been received in time, i.e., check if $C(i,t) \leq endcommphase_i(length((\mathbf{p};i))-1)$. If m(mv,(**p**;i)) has been received in time, i accepts m(mv,(**p**;i)), otherwise i rejects m(mv,(**p**;i)).*
4. *If m(mv,(**p**;i)) is the first valid message of the DCFAP that i receives (and hence, accepts), i starts the first communication phase of its sub-DCFAP at receipt of m(mv,(**p**;i)). Processor i now calculates at which clock values C(i,t') of its processor clock each of the T+1 communication phases of its sub-DCFAP ends. For all h, with $1 \leq h \leq T+1$, it holds that*

$$endcommphase_i(h) = C(i,t) + \sum_{k=1}^{h} L_k$$

Furthermore, processor i is triggered to start its own BAP (BAP_i) belonging to the DCFAP. Hence, processor i acts as the source in BAP_i belonging to the DCFAP and generates and sends a message $m(mv_i,(i;k))$ to every processor $k \in N \setminus \{i\}$. Here, $mv_i = F_i(B(i))$. Processor i accepts a copy of its own message in order to use it in the decision-making process of BAP_i.
5. *If i accepted m(mv,(**p**;i)) and path(m(mv,(**p**;i))) contains fewer than T+2 processors, for every processor h that is not in path(m(mv,(**p**;i))), i signs message m(mv,(**p**;i)) and i relays m(mv,(**p**;i)) to h, immediately after m(mv,(**p**;i)) was received by i. As stated in Section 3.3.3.3., relaying message m(mv,(**p**;i)) to any processor h results in m(mv,(**p**;i;h)) being sent to h.*

All correct processors are assumed to execute the above-described protocol. The behaviour of faulty processors may deviate arbitrarily from the above-described protocol.

Assume a DCFAP based on function sharing is executed in a fully-connected synchronous system *N* consisting of *N* processors (including a source *s*), up to *T* of which may behave maliciously. We assume that for any BAP belonging to the DCFAP, assumptions A3.1 through A3.8 (in Section 3.3.8.), and A4.14 and A4.15 are satisfied, where A4.15 is given by

A4.15. Every processor in the system may have a different view of time and of the start and end of communication phases in the DCFAP. For all *i*, with $1 \leq i \leq T+1$, for any correct processor $p \in N$, let L_i be the length of communication phase *i* of *p*'s sub-DCFAP, as viewed from *p*. Then:

$$L_1 = 2\tau_{max} \cdot (1+\rho)$$

$$L_2 = 2\tau_{max} \cdot (1+\rho)^3 + \Delta \cdot (1+\rho)$$

$$(\forall\, h \in [3,T+1]{::}\ L_h = [L_{h-1} \cdot (1+\rho) + \Delta] \cdot (1+\rho))$$

Analogously as it is done in Section 3.4.1. for authenticated self-synchronizing ICAs, it can be proven that provided that, in a DCFAP, the above-mentioned assumptions hold, then the conditions DCFAP1 and DCFAP2 given in Section 4.1.3.4., are satisfied. The verification of this is left to the reader.

In the above-described DCFAP, it is guaranteed that all correct processors agree on the same set of shares. Since the order in which every correct processor tries the possible combinations of T+1 shares is the same for each correct processor, and each correct processor stops as soon as it has found a correct combination, it is guaranteed that all correct processors arrive at the same decision about the result of encryption of data B with secret cryptographic function F.

Now, consider again the purpose for which the DCFAP is used: encryption of the hash value of a re-created recovered data fragment as part of the process of signing re-created recovered data fragments. For this application, it is sufficient that (provided the number of faulty processors is less than or equal to T) every correct processor produces a correct encrypted hash value. Since the hash value for a certain data fragment is unique, there can only exist one correct encrypted hash value. So, provided that every correct processor produces a correct encrypted hash value, it is automatically guaranteed that all encrypted hash values of all correct processors are equal. Every correct processor is able to produce a correct encrypted hash value, after it has received at least T+1 correct shares. Since it is assumed that there are at least T+1 correct processors in the system, it can easily be guaranteed that all correct processors receive at least T+1 shares by having every correct processor communicate the share it has calculated to all other processors in the system. This leads to a simplified protocol that consists of only one communication phase. The details of this simplified protocol (referred to as simplified DCFAP) are given below.

Initially, for each processor $i \in N$, $endcommphase_i = \infty$

For some (possibly more than one) processor $s \in N$:
Event: *Processor s is triggered to start the simplified DCFAP at time $C(s,t)$.*
Action: *Processor s generates and sends a message $m(mv_s,(s;k))$ to every processor $k \in N \setminus \{s\}$. Here, $mv_s = F_s(B(s))$. Processor s accepts a copy of its own message in order to use it in the decision-making process of its sub-protocol.*
Simultaneously, the source starts the communication phase of its sub-protocol. Furthermore, the source calculates at which clock value $C(s,t')$ of its processor clock the communication phase of its sub-protocol ends. It holds that

$$endcommphase_s = C(s,t) + 2\tau_{max} \cdot (1 + \rho)$$

For each processor $i \in N$:
Event: *$C(i,t) \geq endcommphase_i$*
Action: *For every $k \in N$, processor i decides the value of k's share to be equal to the message value of the messages which i has accepted from k in its sub-protocol, or a default value, if i did not accept such a message. Then, processor i decides the outcome of the simplified DCFAP on basis of the shares (for any $k \in N$) it has calculated. Processor i acts as a combiner in order to calculate $F(B)$, i.e., the result of application of secret cryptographic function F on the commonly available input data B. Provided that at least T+1 shares (for*

any $k \in N$) equal the result of evaluation of shadow function F_k on the commonly available input data B, it is easy to calculate F(B).
The decision taken in processor i is determined as follows. Processor i evaluates the correctness of any combination of T+1 shares in a fixed, a priori agreed order, until the first correct combination is encountered (in this case, this correct combination is taken as i's decision), or until no combination is left anymore (in this case, processor i decides a default value). Any combination X is decided to be correct iff $F^{(-1)}(X) = B$.

Event: *message $mess_j(i)$ (with $1 \le j \le M_i$) received at time C(i,t)*

Action:

1. *Check if $mess_j(i)$ is a valid message, i.e., check if $mess_j(i) \in$ ValidMessages.*
   If invalid($mess_j(i)$) then reject $mess_j(i)$ and abort, i.e., do no perform further actions on $mess_j(i)$.
   If valid($mess_j(i)$), then $mess_j(i)$ = m(mv,($\mathbf{q}$)) of a sub-protocol of some processor s belonging to the simplified DCFAP (with s = first(($\mathbf{q}$)), for some path ($\mathbf{q}$) $\in$ ValidPaths and some mv $\in$ MessageValues. If length(($\mathbf{q}$)) $\neq 2$, or last(($\mathbf{q}$)) $\neq$ i, then reject $mess_j(i)$ and abort, otherwise, $mess_j(i)$ = m(mv,(s;i)) for some $s \in N$.
2. *Check if a valid message $mess_k(i)$ (with $1 \le k < j$) with path($mess_k(i)$) = (s;i) has already been accepted.*
   If such a message has already been accepted, then reject $mess_j(i)$ and abort.
3. *Check if m(mv,(s;i)) has been received in time, i.e., check if C(i,t) $\le$ $endcommphase_i$. If m(mv,(s;i)) has been received in time, i accepts m(mv,(s;i)), otherwise i rejects m(mv,(s;i)).*
4. *If m(mv,(s;i)) is the first valid message of the simplified DCFAP that i receives (and hence, accepts), i starts the communication phase of its sub-protocol at receipt of m(mv,(s;i)). Processor i now calculates at which clock value C(i,t') of its processor clock the communication phase of its sub-protocol ends. It holds that*

$$endcommphase_i = C(i,t) + 2\tau_{max} \cdot (1+\rho)$$

   Furthermore, processor i generates and sends a message m(mv_i,(i;k)) to every processor $k \in N \setminus \{i\}$. Here, $mv_i = F_i(B(i))$. Processor i accepts a copy of its own message in order to use it in the decision-making process of its sub-protocol.

All correct processors are assumed to execute the above-described protocol. The behaviour of faulty processors may deviate arbitrarily from the above-described protocol.

4.4. Comparison of DCFAPs based on a fault-tolerant multisignature scheme with DCFAPs based on function sharing

In this section, we will compare the DCFAPs based on a fault-tolerant version of the multisignature scheme (described in Section 4.2.) with the DCFAPs based on function sharing (described in Section 4.3.)

Both types of DCFAPs are defined for a set *N*, of *N* processors. In both types of DCFAPs, every processor is capable of calculating a share of the secret cryptographic function *F*. The DCFAPs described in Section 4.2. are *interactive* DCFAPs, i.e., the processors in *N* should interact in order to calculate their share, whereas the DCFAPs described in Section 4.3. are *non-interactive* DCFAPs, i.e., any processor in *N* may calculate its share independently of the other processors.

The non-interactive DCFAPs have several advantages over the interactive DCFAPs.

First, in the non-interactive DCFAPs, for $T > 1$, the minimally required number of processors may be smaller than in the interactive DCFAPs[30]. Viz., in the interactive DCFAPs it may be required that $N \geq T \cdot (T+1) + 1$ (for cck-distribution 2), whereas in the non-interactive DCFAPs, $N \geq 2T+1$ always suffices.

Furthermore, for $T > 0$, the non-interactive DCFAPs require less computation overhead than the interactive DCFAPs. The multicast processes of the BAPs that together make up a *non-interactive* DCFAP, consist of T+1 communication phases. For *interactive* DCFAPs, of which the multicast processes of the PCFAPs that together make up the interactive DCFAP contain more than T+1 communication phases, the communication overhead, and hence, the computation overhead is greater than that in a non-interactive DCFAP. However, even in an interactive DCFAP of which the multicast process of all component PCFAPs consist of only T+1 communication phases, the computation overhead may also be greater than in a non-interactive DCFAP, since, in an interactive DCFAP, in every communication phase encryption and decryption of received data fragments take place (abundantly requiring computation time), whereas in a non-interactive DCFAP, encryption and decryption of received data only happen in the last communication phase. If $T > 0$, then the multicast processes of the component PCFAPs (respectively BAPs) of the DCFAPs contain more than one communication phase, and hence, the computation overhead in an interactive DCFAP is greater than in a non-interactive DCFAP. If $T = 0$, then both types of DCFAPs require equal computation overhead.

From the above, we conclude that the non-interactive DCFAPs described in Section 4.3. are to be preferred to the interactive DCFAPs described in Section 4.2.

4.5. Conclusion

In this chapter, distributed cryptographic function application protocols (DCFAPs) have been introduced. DCFAPs can be used in order to store data in a fault-tolerant and secure way on a set of processors, up to *T* of which may behave maliciously.

In a DCFAP, a group of *N* processors, up to *T* of which may behave maliciously, applies a secret cryptographic function *F* on a commonly known piece of data, *B*, while satisfying the following properties:

30. For T=0 or T=1, the minimally required number of processors is equal for both types of DCFAPs.

❑ malicious behaviour of up to *T* processors from *N* does not inhibit application of cryptographic function *F*.
❑ any group of *N*-*T* or more processors from *N* is capable of applying function *F*.
❑ *T* or fewer colluding processors from *N* are unable to compute the secret cryptographic function *F*, or to apply *F* without the help of one or more correct processors.

DCFAPs are based on the idea of giving every processor in the system the capability to calculate a part of the result of application of *F* on *B*, i.e., every processor is capable of calculating its share. DCFAPs are needed in order to sign a re-created recovered data fragment.

Two different types of DCFAPs have been considered:
❑ DCFAPs based on a fault-tolerant version of the multisignature scheme in [Okam88]. These DCFAPs are interactive DCFAPs, i.e., during execution of the DCFAP, the processors interact in order to calculate their share.
❑ DCFAPs based on function sharing. These DCFAPs are non-interactive DCFAPs, i.e., processors may calculate their share independently of the other processors.

The non-interactive DCFAPs are usually to be preferred to the interactive DCFAPs, since, for $T > 1$, the minimally required number of processors is smaller than in the interactive DCFAPs[31], and, moreover, for $T > 0$, the non-interactive DCFAPs require less computation overhead than the interactive DCFAPs[32].

4.6. References

[ArAs70] Ardin, B.W., and Astill, K.N., **Numerical Algorithms**, Addison-Wesley, Reading, Massachusetts, 1970.

[Blak79] Blakley, G.R., Safeguarding cryptographic keys, in: **Proceedings of the AFIPS 1979 National Computer Conference**, New York, Vol. 48, June 1979, pp.313-317.

[Boer96] Boer, W. de, **Applying Cryptography in a Scattered Backup System**, Graduation thesis SPA-96-06, University of Twente, Enschede, May 1996.

[Boyd89] Boyd, C., Digital multisignatures, in: **Cryptography and coding**, Beker, H., and Piper, F., (Eds.), Clarendon Press, 1989, pp.241-246.

[Bree89] Breeding, K.J., **Digital Design Fundamentals**, Prentice Hall International Editions, 1989, ISBN 0-13-211830-0,

[CASD95] Cristian, F., Aghili, H., Strong, R., and Dolev, D., Atomic broadcast: From simple message diffusion to Byzantine agreement, in: **Information and Computation**, Vol.118, 1995, pp.158-179. (An earlier version of this paper appeared in: **Proceedings of the 15th International Symposium on Fault-Tolerant Computing**, Ann Arbor, June 1985, pp.200-206)

[CoBo72] Conte, S.D., and de Boor, C., **Elementary Numerical Analysis**, McGraw-Hill, New York, 1972.

[CrHa89] Croft, R.A., and Harris, S.P., Public-key cryptography and re-usable shared secrets, in: **Cryptography and coding**, Beker, H., and Piper, F., (Eds.), Clarendon Press, 1989, pp.189-201.

[Davi83] Davies, D.W., Applying the RSA digital signature to electric mail, in: **IEEE Computer**, Feb. 1983, pp.55-62.

[DDFY94] De Santis, A., Desmedt, Y., Frankel, Y., and Yung, M., How to Share a Function Securely, in: **Proceedings of the 26th Annual ACM Symposium on the Theory of Computing**,

31. For T=0 or T=1, the minimally required number of processors is equal for both types of DCFAPs.
32. For T=0, both types of DCFAPs require equal computation overhead.

Montréal, Canada, 1994, pp.522-533.

[Denn84] Denning, D.E., Digital signatures with RSA and other public-key cryptosystems, in: **Communications of the ACM**, Vol.27, No.4, April 1984, pp. 388-392.

[Desm88] Desmedt, Y., Society and group oriented cryptography: a new concept, in: **Advances in Cryptology, Proceedings of CRYPTO'87**, Pomerance, C. (Ed.), LNCS 293, Springer-Verlag, 1988, pp.120-127.

[Desm93] Desmedt, Y., Threshold Cryptosystems, in: **Advances in Cryptology - Proceedings of AUSCRYPT'92**, Seberry, J., and Zheng, Y. (Eds.), LNCS 718, Springer-Verlag, 1993, pp.3-14.

[Desm94] Desmedt, Y. G., Threshold Cryptography, in: **European Transactions on Telecommunications**, Vol.5, No.4, July-August 1994, pp.449-457.

[DiHe76] Diffie, W., and Hellman, M.E., New directions in cryptography, in: **IEEE Transactions on Information Theory**, IT 22(6), November 1976, pp. 644-650.

[DiHe79] Diffie, W., and Hellman, M.E., Privacy and authentication: An introduction to cryptography, in: **Proceedings of the IEEE**, Vol. 67, March 1979, pp.397-427.

[Eber93] Eberle, H., A High-Speed DES Implementation for Network Applications, in: Brickell, E.F. (Ed.), **Advances in Cryptology - Proceedings of CRYPTO'92**, LNCS 740, Springer-Verlag, 1993, pp.521-539. Appeared also as: Technical Report No. 90, Digital Systems Research Center, Palo Alto, California, September 1992.

[ElGa85] El Gamal, T., A public key cryptosystem and a signature scheme based on discrete logarithms, in: **IEEE Transactions on Information Theory**, Vol. 31, No.4, July 1985, pp. 469-472.

[FIPS77] Federal Information Processing Standard PUB 46, **Data Encryption Standard**, National Technical Information Service, Springfield, VA, 1977.

[FrDe92] Frankel, Y., and Desmedt, Y., **Parallel reliable threshold multisignature**, Technical Report TR-92-04-02, Department of E.E. and C.S., University of Wisconsin-Milwaukee, April 1992.

[Gong90] Gong, L., **Cryptographic Protocols for Distributed Systems**, Ph.D. thesis, University of Cambridge, 1990.

[Krol91] Krol, Th., **A Generalization of Fault-Tolerance Based on Masking**, Ph.D. thesis, Eindhoven University of Technology, 1991.

[MaSl78] MacWilliams, F.J. and Sloane, N.J.A., **The Theory of Error-Correcting Codes**, Amsterdam, the Netherlands, North Holland, 1978.

[McEl78] McEliece, R.J., A public-key cryptosystem based on algebraic coding theory, **DSN Progress Report 42-44**, Jet Propulsion Laboratory, Pasadena, CA, 1978, pp.114-116.

[Merk78] Merkle, R.C., Secure communications over insecure channels, in: **Communications of the ACM**, Vol.21, No.4, April 1978, pp.294-299.

[Merk82] Merkle, R.C., **Secrecy, Authentication, and Public Key Systems**, UMI Research Press, Michigan, 1982. A revision of 1979 Ph.D. thesis at Stanford University.

[Merk89] Merkle, R.C., One Way Hash Functions and DES, in: **Advances in Cryptology - Proceedings of CRYPTO'89**, LNCS 435, Springer-Verlag, 1989, pp. 428-446.

[NiZu72] Niven, I., and Zuckerman, H.S., **An Introduction to the Theory of Numbers**, Wiley, New York, 1972.

[Okam88] Okamoto, T., A Digital Multisignature Scheme Using Bijective Public-Key Cryptosystems, in: **ACM Transactions on Computer Systems**, Vol.6, No.8, November 1988, pp.432-441.

[PBHS96] Postma, A., Boer, W. de, Helme, A., and Smit, G., **Distributed Encryption and Decryption Algorithms**, Memoranda Informatica 96-20, University of Twente, Enschede, December 1996.

[PoKM97] Postma, A., Krol, Th., and Molenkamp, E., Distributed Cryptographic Function Application Protocols, in: Han, Y., Okamoto, T., and Qing, S. (Eds.), **Information and Communications Security, First International Conference, ICICS'97, Beijing, China, November 1997, Proceedings**, LNCS 1334, Springer-Verlag, Berlin, 1997, ISBN 3-540-63696-X, pp.435-439.

[RiSA78] Rivest, R., Shamir, A., and Adleman, L., A method for obtaining digital signatures and public-key cryptosystems, in: **Communications of the ACM**, Vol.21, 1978, pp.120-126.

[Sham79] Shamir, A., How to share a secret, in: **Communications of the ACM**, Vol.22, No.11, November 1979, pp.612-613.

[Shan49] Shannon, C.E., Communication theory of secrecy systems, in: **Bell System Technical**

Journal, Vol.28, Oct. 1949, pp.656-715.

[SiLL90] Simons, B., Lundelius Welch, J., and Lynch, N., An Overview of Clock Synchronization, in: Simons, B., and Spector, A. (Eds.), **Fault-Tolerant Distributed Computing**, LNCS448, Springer Verlag, Berlin, 1990, ISBN 3-540-97385-0, pp.84-96.

[Simm92] Simmons, G.J. (Ed.), **Contemporary Cryptology, The Science of Information Integrity**, IEEE Press, New York, 1992.

[VaOo89] Vanstone, S.A., and van Oorschot, P.C., **An Introduction to Error Correcting Codes with Applications**, Kluwer Academic Publishers, Boston, 1989.

[Wake90] Wakerly, J.F., **Digital Design Principles and Practices**, Prentice Hall International Editions, 1990, ISBN 0-13-212879-9.

[Wilk72] Wilkes, M.V., **Time-Sharing Computer Systems**, New York, American Elsevier, 1972, p.91.

[Wint84] Winternitz, R.S., A Secure One-Way Hash Function Built from DES, in: **Proceedings of the 1984 IEEE Symposium on Security and Privacy**, Oakland, CA, 1984, pp.88-90.

CHAPTER 5

A Byzantine Fault-Tolerant Distributed Diagnosis Algorithm

ANTEATER :
“Most of my clients suffer from some sort of speech impairment. You know, colonies which have to grope for words in everyday situations. It can be quite tragic. I attempt to remedy the situation by, uhh - removing - the defective part of the colony. These operations are sometimes quite involved, and of course years of study are required before one can perform them.”

Douglas Hofstadter - Gödel, Escher, Bach: An Eternal Golden Braid

Abstract

In general, offering a fault-tolerant service boils down to the execution of replicas of a service process on different processors in a distributed system. The service is fault-tolerant in such a way, that, even if some of the processors on which a replica of the service resides, behave maliciously, the service is still performed correctly. To be able to guarantee the correctness of a fault-tolerant service despite the presence of maliciously functioning processors, it is of key importance that all faulty processors are timely removed from this service. Faulty processors are detected by tests performed by the processors offering the service. In practice, tests always have an imperfect fault coverage. In this chapter, a distributed diagnosis algorithm with imperfect tests is described, by means of which all detectably faulty processors are removed from a fault-tolerant service. This may, however, inevitably, imply the removal of a number of correctly functioning processors from the service too. The maximum number of correctly functioning processors removed from the service by the algorithm is calculated. Finally, the minimally required number of processors needed in a fault-tolerant service to perform this diagnosis algorithm is given.

5.1. Introduction

Reliable computer systems have become indispensable in today's society. The services provided by these systems are usually implemented in a fault-tolerant way as a set of replicated processes, where every replica of a process runs on a different processor. These fault-tolerant services satisfy their specification as long as the number of processors that concurrently fail is less than a certain predefined maximum. To avoid inconsistency within such a fault-tolerant service, it is highly important that faulty processors are excluded from the fault-tolerant service as soon as possible.

System-level fault diagnosis algorithms are a convenient way to determine faulty processors within a set of processors on which a fault-tolerant service resides. After the basic work by Preparata, Metze and Chien [PrMC67], called the PMC model, a vast amount of system-level fault diagnosis algorithms have appeared in literature. In [BaMD93], several extensions to the PMC model are given.

Many diagnosis algorithms assume permanently faulty processors and tests having a 100% fault coverage [BaMa91, BiGN90, Jalo94, PrMC67]. In reality, tests do not have a 100% fault coverage. Therefore, in literature, algorithms have appeared that use a probabilistic technique to cope with imperfect tests [BlSM92, LeSh90, LeSh93, LeSh94]. On basis of the test results, these so-called probabilistic diagnosis algorithms find the set of faulty processors that is most likely to have produced the test results. As the number of processors in the system grows to infinity the percentage of correct diagnoses approaches 100%. Adaptations to the probabilistic algorithms described in [ChBE93, Kime70, MaHa76] assume a restricted fault model. For every possible set of test results within such a restricted fault model, we know the corresponding set of faulty processors.

To achieve a sufficiently high reliability, it may be necessary to exclude all faulty processors from a fault-tolerant service. If the service processors suffer from Byzantine failures and the performed tests do not have a 100% fault coverage, the algorithms mentioned above are not applicable.

This chapter describes a new system-level distributed diagnosis algorithm, by means of which all faulty processors are removed from the system, and which is based on the assumptions that processors may fail in an arbitrary way and that tests need not have a 100% fault coverage. Byzantine faults can be handled by having a worst case approach in a distributed environment. In the cases in which the test results are indeterminate, all possibly faulty processors are excluded from the service. This implies that in some cases, a number of correctly functioning processors must also be excluded from the service.

In the following section, we will describe our new diagnosis algorithm. Then, we will calculate the maximum number of correctly functioning processors that may be removed by the algorithm. Finally, we calculate the minimum number of processors needed in order to be able to perform the described diagnosis algorithm as a function of the maximum number of faulty processors that is tolerated.

The algorithms described in this chapter have been published previously in [PoHK96]. An earlier version of these algorithms can be found in [Hart95].

5.2. A distributed diagnosis algorithm with imperfect tests

In this section, we will describe our distributed diagnosis algorithm with imperfect tests. This diagnosis algorithm is executed by every processor of the fault-tolerant service.

Let N be the set of all processors in the fault-tolerant service. Every processor in N executes this diagnosis algorithm, independent of the other processors in N.

It is assumed that, *before the start* of the diagnosis algorithm, every processor in N has tested every other processor in N , and that these test results are available in every processor. We assume that the test results have been distributed among the processors by means of Byzantine Agreement Protocols[1], so that every processor has its own copy of the test results before the start of the algorithm[2], and, moreover, all correct processors agree on the test results.

The diagnosis algorithm consists of the following two steps:

DA1. In the first step, on basis of the test results, the processor that executes the diagnosis algorithm tries to determine which processors are faulty and which processors are correct. Processors that are determined to function correctly on basis of the test results are classified in the so-called ***correct-set*** C. Processors that are guaranteed to be faulty are classified in the so-called ***fault-set*** F. Processors for which, on basis of the test results, it cannot be determined whether they are correct or faulty, are also classified in the fault-set.

Hence, the first step results in the creation of the correct-set C, which contains the identifiers of all processors that are determined to function correctly (on basis of the test results), and the fault-set F, which contains the identifiers of the other processors, being all the faulty processors and possibly, some correctly functioning processors also. It holds that $F \cup C = N$.

DA2. Then, all processors in the fault-set F are removed from the service. Since we know that all remaining processors in the service (i.e., the processors in the correct-set C) function correctly, it is possible to, one by one, adopt new correct processors in the service, if each remaining processor in the service agrees upon the correctness of the new processors (by testing them).

In this chapter, we will only concentrate on step DA1 of the diagnosis algorithm, i.e., the classification of all processors from N in either the correct-set C or the fault-set F on basis of the test results.

The test results can be denoted in a matrix M, in which the row indices represent the testing processors and the column indices represent the tested processors. An entry $M_{i,j}$ (with $i,j \in N$) of the matrix represents a test result and can have the value C or F. $M_{i,j} = C$ indicates that the test was successful and j (the tested processor) is considered correct by i (the testing processor). $M_{i,j} = F$ indicates that the test failed, and j (the tested processor) is considered faulty by i (the testing processor).

In step DA1 of the diagnosis algorithm, on basis of the test results, all processors from N are classified either in the correct-set or in the fault-set. This classification is done in the following two steps:

1. See Section 3.1.3.
2. If, due to link failures the system is split into disjoint parts, the test results cannot be obtained, and the diagnosis algorithm cannot be executed. It is impossible to simply assume that a processor from which no test results are obtained is faulty (respectively correct). Then, e.g., assumption A5.1 (respectively A5.2) as formulated in Section 5.2.1. may not hold.

CA1. First, by means of applying a number of so-called ***reduction rules*** (given in Section 5.2.2.), all processors that are guaranteed to be correct are classified in the correct-set, and the processors that are guaranteed to be faulty are put in the fault-set.

CA2. After no reduction rules can be applied anymore, the processors that have not yet been classified are classified in the fault-set. This is done in order to guarantee that all possibly faulty processors are classified in the fault-set, and hence, will be removed from the fault-tolerant service.

The rest of this section will be devoted to step DA1 of the diagnosis algorithm.

First, in Section 5.2.1., the required assumptions will be stated. Then, in Section 5.2.2., we present the reduction rules applied in step CA1. It is interesting to notice that it can be proved that the reduction rules stated in Section 5.2.2. follow from the assumptions in Section 5.2.1. Section 5.2.3. contains a brief investigation of step CA2. Finally, Section 5.2.4. illustrates the use of the reduction rules by means of several examples.

5.2.1. Assumptions

We assume a completely connected network of processors with perfect communication links. In practice, the effect of a link failure is indistinguishable from a failure in the sending processor. For simplicity, we assume that every communication failure is caused by a processor failure, and assume that link failures do not occur. We assume that in the network, at most T processors are faulty during an execution of the diagnosis algorithm. Faulty processors may exhibit Byzantine failures, i.e., they may fail in an arbitrary way. For a processor $n \in N$, the predicate *faulty*(n) means that n is faulty, whereas *correct*(n) means that n is correct. The assumptions concerning the processors and their test results, are:

A5.1. $\forall\, a,b \in N : (\mathit{correct}(a) \wedge \mathit{correct}(b)) \Rightarrow M_{a,b} = C$
i.e., if a processor a and b are both correct, then a has a positive test result for b

A5.2. $\forall\, a \in N : \mathit{faulty}(a) \Rightarrow (\exists\, b \in N : \mathit{correct}(b) \wedge (M_{a,b} = F \vee M_{b,a} = F))$
i.e., if a processor a is faulty, there exists a correct processor for which a has negative test results or there exists a correct processor which has negative test results for a.

A5.3. $\forall\, a \in N : (\mathit{faulty}(a) \Rightarrow \neg\, \mathit{correct}(a)) \wedge (\neg\, \mathit{correct}(a) \Rightarrow \mathit{faulty}(a))$
i.e., a processor is either faulty or correct

A5.4. $\forall\, a,b \in N : M_{a,b} \in \{F, C\}$
i.e., test results are either positive or negative

A5.5. $|\, \{ a \in N : \mathit{faulty}(a)\} \,| \le T$
i.e., the number of faulty processors is less than or equal to T

A5.2 expresses our assumption that faulty processors exhibit detectably faulty behaviour, i.e., it is impossible that a faulty processor n has positive test results for every correct processor, and every correct processor has positive test results for n.

5.2.2. Reduction rules

Using the assumptions given in Section 5.2.1., for a given set of test results of a set N

of processors, we are able to determine for a number of processors of N whether they should be put in the correct-set C or in the fault-set F. This is done by a repeated application of the following reduction rules on the processors in set N until no more processors can be classified anymore.

The reduction rules are as follows:

R5.1. $\forall\, n \in N - F - C : (M_{n,n} = F \Rightarrow n \in F)$
i.e., if a processor has negative test results for itself, it is put in the fault-set.

R5.2. $\forall\, n \in N - F - C : |\{\, i \in N - F \mid M_{i,n} = F \vee M_{n,i} = F \,\}| > (T - |F|) \Rightarrow n \in F$
i.e., if the number of processors that are not in the fault-set and for which a processor n has negative test results or which have negative test results for n exceeds T minus the number of processors in F, then n is put in the fault-set.

R5.3. $\forall\, n \in N - F - C : (\forall\, m \in N - F : M_{m,n} = C \wedge M_{n,m} = C) \Rightarrow n \in C$
i.e., if all processors that are not in the fault-set have positive test results for a certain processor n, and n has positive test results for all processors not in the fault-set, we assume that this processor n is correct, and it is put in the correct-set.

By means of application of the above three reduction rules, it is possible to classify several processors from N in the fault-set and the correct-set, respectively. In general, a set N ‘ of processors will remain that cannot be classified in either the fault-set or the correct-set by means of application of any of the rules R5.1 through R5.3. This set N ’ will contain at most $T' = T - |F|$ faulty processors.

Sometimes, it may be possible to classify some more processors, by using a fourth reduction rule R5.4. Before this rule is presented, we first give an outline of the strategy that is used and we introduce several definitions.

We suppose one of the processors of N ‘ , say n, is correct. Then, by assumption A5.1., all processors m for which $M_{n,m} = F$ or $M_{m,n} = F$, must be faulty. Now, we try to find all subsets P of N ‘, with $n \in P$, such that it is possible, according to assumptions A5.1. through A5.5., that all processors in P are correct, while all processors in N ‘ - P are faulty. I.e., for set P, it must hold that

$$n \in P \wedge (\forall\, a,b \in P : M_{a,b} = C)$$

and for N ‘ - P

$$|N' - P| \leq T'$$

In general, many solutions for P can be found. However, there may be processors $q \in N$ ‘ for which such a set P does not exist. Any processor q for which such a set P does not exist, must thus be faulty, and hence, q is put in the fault-set by reduction rule R5.4.

Before we present reduction rule R5.4., we first introduce several definitions.

For any processor $n \in N$ ‘, we define its set of ***suspicious processors*** as the set of processors which must be faulty, if n would be correct (i.e., the set of all processors m for which $M_{n,m} = F$ or $M_{m,n} = F$). Thus, for every processor $n \in N$ ‘, with $N' = N - F - C$, the set $S(n, N')$ of suspicious processors is defined as follows:

$$S(n, N') = \{\, m \in N' \mid M_{m,n} = F \vee M_{n,m} = F \,\}$$

For a set $X \subset N'$, the set $S(X, N')$ of suspicious processors for set X is defined as follows:

$$S(X, N') = \bigcup_{i \in X} S(i, N')$$

We will define a ***presupposed correct-set for a processor n within set*** N' as a set of processors which, if we assume these processors correct, results in a consistent solution (i.e., a classification of all processors which satisfies assumptions A5.1. through A5.4. stated in Section 5.2.1.). For a presupposed correct-set P for n within N', we require that

$$(n \in P) \wedge (S(P, N') \cap P = \emptyset)$$

Clearly, for certain $n \in N'$, more than one presupposed correct-set may exist. Therefore, we define the set $PS(n, N')$ of all presupposed correct-sets of n within N' as follows:

$$PS(n, N') = \{ P \mid n \in P \wedge (S(P, N') \cap P = \emptyset)\}$$

Now, we are able to formulate reduction rule R5.4:

R5.4. $\forall\, n \in N - F - C : (\neg (\exists\, P \in PS(n, N - F - C) : |\, N - F - C - P \,| \leq T - |\, F \,|)) \Rightarrow n \in F$

i.e., if for a certain processor n no presupposed correct-set P can be found, such that the number of processors in $N - F - C - P$ is less than or equal to $T - |\, F \,|$, then n is put in the fault-set.

It is interesting to remark that reduction rule R5.2. is contained in reduction rule R5.4., i.e., if a processor can be classified by means of reduction rule R5.2., it can also be classified by means of R5.4.

By repeated application of reduction rules R5.1 through R5.4 we can classify several processors from N either in the fault-set or the correct-set. In Appendix 5.1, we prove that the predicates:

$$\forall\, n \in F : \mathit{faulty}(n)$$
$$\forall\, n \in C : \mathit{correct}(n)$$

are invariant under application of the reduction rules R5.1 through R5.4. This means that, by the reduction rules only faulty processors are put in the fault-set and only correct processors are put in the correct-set.

Provided that reduction rule R5.1 is first applied repeatedly, the other reduction rules may be applied in any order. Furthermore, the rules are consistent, i.e., it is impossible to classify a processor both in the fault-set and in the correct-set. A proof of this can be found in Appendix 5.2.

5.2.3. Classification of the remaining processors after having applied the reduction rules

If for a remaining set of processors none of the reduction rules in the previous section can be applied anymore, then the remaining processors satisfy the conjunction of the negations of the premises of the reduction rules. The remaining processors form a set $N' = N - F - C$ of processors with so-called ***indeterminate test results***. The processors in N' satisfy:

$$\forall\, n \in N' : \quad ((M_{n,n} = C) \wedge (\,|\,\{\, i \in N - F \mid M_{i,n} = F \vee M_{n,i} = F\}\,| \le (\,T - |\,F\,|\,) \wedge$$
$$(\,\exists\, m \in N' : M_{m,n} = F \vee M_{n,m} = F) \wedge$$
$$(\exists\, P \in PS(n, N') : |\, N' - P \,| \le T - |\,F\,|\,))$$

Notice that, if *N* ‘ is non-empty, one or more processors in *N* ‘ must be faulty. This can be seen as follows. From the definition of indeterminate test results given above, we conclude that every processor in *N* ‘ has negative test results for another processor in *N* ‘, or there is another processor in *N* ‘ which has negative test results for it (viz. $\forall\, n \in N'$: $(\,\exists\, m \in N' : M_{m,n} = F \vee M_{n,m} = F))$). Let *a* and *b* be processors in *N* ‘. Assume that processor *a* has negative test results for processor *b*. Then, by assumption A5.1, either processor *a* or processor *b* must be faulty.

In order to exclude all faulty processors from the fault-tolerant service, all processors from *N* ‘ are put in the fault-set *F*. Then, all processors from *N* have been classified and the algorithm ends. Since *N* contains a finite number of processors, the algorithm always ends.

We will illustrate the definition of indeterminate test results by means of an example:

Example 5.1.
Assume a set *N* contains two processors 1 and 2. Assume furthermore that the maximum number of faulty processors *T* = 1. Processor 1 has negative test results for processor 2 and positive test results for itself. Processor 2 has negative test results for processor 1 and positive test results for itself. For this example, matrix *M* is defined as follows:

$$M = \begin{bmatrix} C & F \\ F & C \end{bmatrix}$$

Reduction rule R5.1 through R5.4 can not be applied to these test results. The test results of processor 1 and processor 2 are indeterminate. So processor 1 and 2 are put in the fault-set and the algorithm ends. ❏

5.2.4. Examples of the use of the reduction rules

We will now explain the use of the reduction rules by means of three examples.

Example 5.2.
Assume a service contains four processors 1,2,3, and 4. Assume furthermore that *T* = 2. The test results of the processors 1, 2, 3, and 4 are given in the matrix below. The

$$M = \begin{bmatrix} C & F & F & C \\ C & C & C & C \\ C & C & C & F \\ F & C & C & C \end{bmatrix}$$

algorithm proceeds as follows. Initially, $F = \emptyset$ and $C = \emptyset$. Reduction rule R5.2 can be applied to processor 1. Since $M_{1,2} = M_{1,3} = M_{4,1} = F$, processor 1 can be put in the fault-set. Then we apply reduction rule R5.3 to put processor 2 in the correct-set. Now, we cannot apply reduction rules anymore. so processor 3 and 4 are put in the fault-set.

Notice that processor 2 is assumed to be correct, although processor 2 has a positive test result for processors 1, 3, and 4. This is possible, since we assumed that tests do not have a perfect coverage. Notice that it is possible that by application of the diagnosis algorithm one or more correctly functioning processors are put in the fault-set (In this example, finally, $F = \{1,3,4\}$ while $T = 2$). ❑

Example 5.3.
Assume a service contains five processors 1,2,3,4, and 5. Assume furthermore that $T = 2$. The test results for the processors are given in the matrix below. By applying reduc-

$$M = \begin{bmatrix} C & C & C & F & C \\ F & F & F & C & C \\ C & F & C & C & C \\ F & C & C & C & C \\ C & F & C & C & C \end{bmatrix}$$

tion rule R5.1, processor 2 is put in the fault set. It is not possible to apply rule R5.2 to the remaining test results. By rule R5.3, processor 3 and 5 are put in the correct-set. The remaining processors (processor 1 and 4) show indeterminate test results. Both processor 1 and processor 4 are put in the fault-set. Thus, our algorithm concludes with F consisting of processor 1, 2, and 4, and C consisting of processor 3 and 5. ❑

Example 5.4.
Assume a service contains five processors 1,2,3,4, and 5. Assume furthermore that $T =$

$$M = \begin{bmatrix} C & F & F & C & C \\ F & C & C & C & C \\ F & C & C & C & C \\ C & C & C & C & F \\ C & C & C & F & C \end{bmatrix}$$

2. The test results for the processors are given in the matrix above. Reduction rule R5.1 through R5.3 can not be applied to these results. We can, however, apply reduction rule R5.4 to processor 1. The possible presupposed correct-sets for processor 1 are {1,4} and {1,5}. Both have a number of suspicious processors greater than T, so processor 1 must be put in the fault-set. By reduction rule R5.3, then, processor 2 and 3 can be put in the correct-set. Processor 4 and 5 have indeterminate test results, therefore, they are both put in the fault-set and the algorithm ends with F consisting of processor 1,4, and 5, and C consisting of processor 2 and 3. ❑

5.3. Determining the maximum number of correct processors removed from the service by the diagnosis algorithm

Putting all processors from N' in the fault-set may result in the removal of correct processors from a service. In this section, we will derive an upper bound for the number of correct processors that may be removed from a service by using the above-described diagnosis algorithm. We first give some definitions we will use in the sequel.

N	the total number of processors in the service.

T	the maximum number of faulty processors in the service
S	the actual number of faulty processors ($S \leq T$)
b	the number of faulty processors detected by application of reduction rule R5.1 through R5.4
a	the number of correct processors detected by application of reduction rule R5.1 through R5.4

We will denote N' as the set of indeterminate test results. For this remaining set N' we use the following definitions:

N'	$= N - (a + b)$	(the number of processors in set N')
T'	$= T - b$	(the maximum number of faulty processors in N')
S'	$= S - b$	(the actual number of faulty processors in N')
G'	$= N' - S'$	(the actual number of correct processors in N')

Set N' contains G' correct processors, all of which are put in the fault-set. In general, we do not know G'. We can only derive an upper bound to the number of correctly functioning processors put in the fault-set. The following lemma is used for this purpose.

For every processor $n \in N'$, one or more presupposed correct-sets $P \in PS(n,N')$ with $|N' - P| \leq T - |F|$ exist. Notice that such a presupposed correct-set with $|N' - P| \leq T - |F|$ does not necessarily contain correct processors only. (In order to see this, recall example 5.1. from Section 5.2.2. In this example, two such presupposed correct-sets can be found, viz. $P_1 = \{1\}$ and $P_2 = \{2\}$. Since $T - |F| = 1$, one of the processors 1 and 2 may be faulty. In that case there is a presupposed correct-set containing a faulty processor).

In Lemma 5.1.1., we will show that if for the processors in N', presupposed correct-sets with $|N' - P| \leq T - |F|$ exist such that every presupposed correct-set contains at most one faulty processor, then the number of correct processors present in the remaining sets of processors is maximal.

LEMMA 5.1.1.
If for every processor $n \in N'$, a presupposed correct-set $P \in PS(n,N')$ with $|N' - P| \leq T - |F|$ exists such that it does not contain more than one faulty processor, the number of correctly functioning processors present in the sets $N' - P$ of processors, is maximal.

Proof:
Assume we have a set N' of processors with indeterminate test results. From the definition of indeterminate test results we know that, for every processor $n \in N'$, there exists a presupposed correct-set $P \in PS(n,N')$ with $S(P, N') \cap P = \varnothing$, and $|N' - P| \leq T - |F|$, and $S(P, N') \cup P \subset N'$ (Notice that, simply selecting $P = \{n\}$ suffices.) Thus:

$$\forall n \in N': \exists P \in \{ P \mid S(P, N') \cap P = \varnothing \wedge S(P, N') \cup P \subset N' \wedge |N' - P| \leq T - |F| \} \wedge n \in P$$

We will now show that for every processor $n \in N'$, there exists a presupposed correct-set $P \in PS(n,N')$ with $S(P, N') \cap P = \varnothing$, and $|N' - P| \leq T - |F|$, and $S(P, N') \cup P =$

N'.

Let P be a presupposed correct-set of some processor n (i.e., $P \in PS(n,N')$), and $S(P, N')$ be the set of suspicious processors for P. Assume furthermore that $S(P, N') \cap P = \emptyset$, and $|N' - P| \le T - |F|$, and $S(P, N') \cup P \subset N'$. Recall that $S(P, N')$ has been defined by

$$\forall\, k : k \in S(P, N') \equiv (\exists\, m \in P : M_{n,m} = F \vee M_{m,n} = F)$$

Suppose a processor $x \in N'$ exists with $x \notin S(P, N') \cup P$ (i.e., x is neither in P nor in $S(P,N')$). Then, from the definition of $S(P, N')$ follows that

$$\neg\, (\exists\, m \in P : M_{x,m} = F \vee M_{m,x} = F)$$

or

$$(\forall\, m \in P : M_{m,x} = C \wedge M_{x,m} = C)$$

hence, x can be added to P.

So, any processor $y \in N'$ with $y \notin S(P, N') \cup P$ can always be added to P and thus, for each processor $n \in N'$, a presupposed correct-set $P \in PS(n, N')$ can be found, such that $S(P, N') \cap P = \emptyset$, and $|N' - P| \le T - |F|$, and $S(P, N') \cup P = N'$.

For this set P, it holds that $|S(P, N')| \le T - |F|$ (since $S(P, N') = N' - P$ and $|N' - P| \le T - |F|$) . Thus:

$$\forall n \in N' : \exists\, P \in \{ P \mid S(P, N') \cap P = \emptyset \wedge S(P, N') \cup P = N' \wedge |S(P, N')| \le T - |F| \} \wedge n \in P$$

We define the set of all presupposed correct-sets P within N' for which $S(P, N') \cup P = N'$ and $S(P, N') \cap P = \emptyset$ and $|S(P, N')| \le T - |F|$ as $CPS(N')$. Thus:

$$CPS(N') = \{ P \mid S(P, N') \cap P = \emptyset \wedge S(P, N') \cup P = N' \wedge |S(P, N')| \le T - |F| \}$$

From the definition of N' and the definition of $S(P, N')$ we know that:

$$\forall\, n \in N' : \exists\, m \in N' : n \in S(m, N')$$

i.e., every processor in N' is element in a set of suspicious processors for another processor in N' within N'.

Thus, for every correct processor y in N', there exists a presupposed correct-set $P \in CPS(N')$ with $y \in S(P,N')$. An upper bound to the maximal number of correct processors put in the fault-set, can therefore be given by

$$upperbound = \left| \bigcup_{P \in CPS(N')} S(P, N') \right|$$

which is equal to $|N'|$.

Notice, however, that for a processor $x \in N'$, in every presupposed correct-set $P \in CPS(N')$, for which $x \in P$, the set of suspicious processors $S(x, N')$ is the same. If processor x is correct, then, by assumption A5.1, its set of suspicious processors $S(x, N')$ contains no other correct processors.

Now, since all processors in N' are element in a set of suspicious processors $S(x, N')$

for some $x \in N'$, and only the sets of suspicious processors $S(x, N')$ for faulty processors x contain correct processors, an upper bound to the maximal number of correct processors put in the fault-set, is also given by

$$upperbound = \left| \bigcup_{x \in N' \wedge faulty(x)} S(x, N') \right|$$

We will calculate the maximal number of correct processors that may be put in the fault-set in the case that X ($X > 1$) presupposed correct-sets from $CPS(N')$ contain only one faulty processor, and also in the case that one presupposed correct-set from $CPS(N')$ contains X faulty processors.

The case: X presupposed correct-sets containing one faulty processor
Assume a presupposed correct-set $P \in CPS(N')$ contains several processors, of which at most one may be faulty. Then, the other $(S' - 1)$ faulty processors must be contained in $S(P, N')$. Furthermore, we know that $|S(P, N')| \leq T'$. Thus, the set of suspicious processors $S(P, N')$ cannot contain more than $T' - (S' - 1)$ correct processors. So, by selecting a presupposed correct-set with one faulty processor, a maximum of $T' - (S' - 1)$ correct processors can be put in the fault-set. So by X of these presupposed correct-sets, $X \cdot (T' - (S' - 1))$ correct processors can be put in the fault-set.

The case: one presupposed correct-set containing X faulty processors
If a presupposed correct-set $P \in CPS(N')$ contains X faulty processors ($X \geq 1$), the remaining set of processors $N' - P$ contains at most $T' - (S' - X)$ correct processors.

Provided that $T' \geq S'$ and $X \geq 1$, it can be derived that $X \cdot (T' - (S' - 1))$ is always greater than or equal to $T' - (S' - X)$. It follows then, that the number of correctly functioning processors being put in the fault-set is maximal, if each presupposed correct-set contains only one faulty processor. ❏

We will now use the result of Lemma 5.1.1. in the following theorem.

THEOREM 5.1.
In the diagnosis algorithm, the number of correct processors accused of being faulty, is always less than or equal to $min[G', S' \cdot (T' - (S' - 1))]$ [3].

Proof:
From assumption A5.1, we know that a correct processor does not accuse other correct processors of being faulty. So a presupposed correct-set containing only correct processors only accuses faulty processors of being faulty. Thus, in order to calculate the maximum number of correct processors accused of being faulty, we need only consider presupposed correct-sets containing one or more faulty processors.

From Lemma 5.1.1., we know that the number of correctly functioning processors accused of being faulty is maximal, if each presupposed correct-set contains at most one faulty processor. The number of faulty processors in N' is S'. Thus at most S' of these presupposed correct-sets can be found. From the foregoing, we know that these S' presupposed correct-sets can accuse a maximum of $S' \cdot (T' - (S' - 1))$ correct proc-

3. For any pair of values p and q, $min[p,q]$ is defined as the minimum of p and q.

essors of being faulty.

Of course, the number of correct processors accused of being faulty can never become greater than the total number G' of correct processors in N'. So, in the diagnosis algorithm, the number of correct processors accused of being faulty, is always less than or equal to

$$min[\, G', S' \cdot (T' - (S' - 1))]. \quad ❏$$

5.4. Determining the maximum number of processors in the service that are accused of being faulty

If we assume that $G' \geq S' \cdot (T' - (S' - 1))$, we find the maximal number of *correct* processors marked faulty, which is equal to $S' \cdot (T' - (S' - 1))$. In the table below, for some practical values of S' and T', we have calculated the maximal number of correct processors accused of being faulty.

	$S' = 1$	$S' = 2$	$S' = 3$	$S' = 4$
$T' = 5$	5	8	9	8
$T' = 4$	4	6	6	4
$T' = 3$	3	4	3	-
$T' = 2$	2	2	-	-
$T' = 1$	1	-	-	-

The maximum number of *faulty* processors marked faulty in the algorithm is equal to S. By definition, $S = S' + T - T'$ (because $S' = S - b$, and $T' = T - b$). The maximum number of (either faulty or correct) processors marked faulty in the algorithm is the sum of the maximum number of correct processors marked faulty and the maximum number of faulty processors marked faulty, and is equal to $S' \cdot (T' - (S' - 1)) + S' + T - T'$.

By differentiating this function on S', we can find the value for S' for which the number of correct processors accused of being faulty, is maximal.

$$f(S') = S' \cdot (T' - (S' - 1)) + S' + T - T' = -S'^{2} + (T' + 2) \cdot S' + T - T'$$

$f(S')$ is maximal for $S' = 1/2 \cdot (T' + 2)$.

For *even* T', the maximum value for $f(S')$ is indeed reached at $S' = 1/2 \cdot (T' + 2)$. Since S' is always an integer value, for *odd* T', the maximum value for $f(S')$ is reached at $S' = 1/2 \cdot (T' + 1)$ and $S' = 1/2 \cdot (T' + 3)$ respectively.

If we calculate the value of $f(S')$ for these values of S', in all cases, we find that the maximum number of processors marked faulty is $\lfloor 1/4 \cdot T'^{2} + T + 1 \rfloor$.(For any positive real value r, $\lfloor r \rfloor$ is defined as the smallest integer value less than or equal to r.). For fixed value of T, the number of processors marked faulty increases with increasing T'. The value of T' is limited by the requirement $T \geq T'$. So, for any value of T, the maximum number of processors marked faulty is found by selecting $T' = T$ in $\lfloor 1/4 \cdot T'^{2} + T$

$+ 1\rfloor$. In the table below, for some practical values of *T*, the maximum number of processors marked faulty is given.

T	1	2	3	4	5
#processors marked faulty $= \lfloor 1/4 \cdot T^2 + T + 1 \rfloor$	2	4	6	9	12

5.5. Determining the minimally required number of processors in the service

After having performed the above-described diagnosis algorithm, the correct processors have a consistent view of the set of processors that must be removed from the service. The only way in which the correct processors can exclude this set of possibly faulty processors is by creating some secret information which is kept within the set of correct processors. This secret information is the new identification of the service to the clients. The remaining processors must form a majority in order to be able to convince clients who use the service that the identification of the service has changed. So the number of processors required in the service must be more than twice the number of processors marked faulty by the diagnosis algorithm. In the table below, for some values of *T*, we have calculated *N*, being the minimally required number of processors in the service.

T	1	2	3	4	5
N	5	9	13	19	25

5.6. Conclusion

In this chapter, we have described a new diagnosis algorithm, by means of which we can guarantee that all processors behaving in a faulty way are removed from a fault-tolerant service. Timely removing all faulty processors is very important in order to be able to guarantee the correctness of a fault-tolerant service despite the presence of up to *T* maliciously functioning processors. Our algorithm is based on the realistic assumption of imperfect tests, therefore, indeterminate test results may occur. In this case, in order to be able to exclude all faulty processors from the service, it may be inevitable, that some correctly functioning processors from the fault-tolerant service are also removed. We have derived an upper bound on the number of correctly functioning processors that may be removed from the fault-tolerant service by execution of the algorithm. In order to be able to tolerate up to *T* maliciously functioning processors, the minimally required number of processors in the service is $2 \cdot \lfloor 1/4 \cdot T^2 + T + 1 \rfloor + 1$.

5.7. References

[BaMa91] Bauch, A., and Maehle, E., Self-Diagnosis, Reconfiguration and Recovery in the Dynamical Reconfigurable Multiprocessor System DAMP, in: **Fault-tolerant computing systems: tests, diagnosis, fault-treatment: 5th International GI/ITG/GMA Conference Nürnberg, September 25-27, 1991: Proceedings**, Dal Cin, M., and Hohl, W. (Eds.), Springer-

Verlag, Berlin, 1991, ISBN 3 540 54545 x, pp. 18-29.

[BaMD93] Barborak, M., Malek, M., and Dahbura, A., The Consensus Problem in Fault-Tolerant Computing, in: **ACM Computing Surveys**, Vol. 25, No. 2, June 1993, pp.171-220.

[BiGN90] Bianchini, R., Goodwin, R., and Nydick, D.S., Practical Application and Implementation of Distributed System-Level Diagnosis Theory, in: **Fault-tolerant computing: the twentieth international symposium (FTCS-20)**, IEEE Computer Society Press, Los Alamitos, California, 1990, ISBN 08186 2051 x, pp. 332-339.

[BlSM92] Blough, D.M., Sullivan, G.F., and Mason G.M. Intermittent Fault Diagnosis in Multiprocessor Systems, in: **IEEE Transactions on Computers**, Vol. 41, No. 11, Nov. 1992, pp.1430-1441.

[ChBE93] Chen, Y., Bücken, W., and Echtle, K., Efficient Algorithms for System Diagnosis with Both Processor and Comparator Faults, in: **IEEE Transactions on Parallel and Distributed Systems**, Vol. 4, No. 4, April 1993, pp.371-381.

[Hart95] Hartman, G., **Membership Management of the Scattered Backup System**, Graduation thesis SPA-95-16, University of Twente, Enschede, October 1995.

[Jalo94] Jalote, P., **Fault Tolerance in Distributed Systems**, Prentice Hall, New Jersey, 1994, pp.115-125.

[Kime70] Kime, C.R., An Analysis Model for Digital System Diagnosis, in: **IEEE Transactions on Computers**, Vol. C-19, No. 11, Nov. 1970, pp.1063-1073.

[LeSh90] Lee, S., Shin, K.G., Optimal Multiple Syndrome Probabilistic Diagnosis,in: **Fault-tolerant computing: the twentieth international symposium (FTCS-20)**, IEEE Computer Society Press, Los Alamitos, California, 1990, ISBN 0 8186 2051 x, pp. 324-331.

[LeSh93] Lee, S., Shin, K.G., Optimal and Efficient Probabilistic Distributed Diagnosis Schemes, in: **IEEE Transactions on Computers**, Vol. 42, No. 7, July 1993, pp.882-886.

[LeSh94] Lee, S., Shin, K.G., On Probabilistic Diagnosis of Multiprocessor Systems Using Multiple Syndromes, in: **IEEE Transactions on Parallel and Distributed Systems**, Vol. 5, No. 6, June 1994, pp.630-638.

[MaHa76] Maheshwari, S.N., Hakimi, S.L., On Models for Diagnosable Systems and Probabilistic Fault Diagnosis, in: **IEEE Transactions on Computers**, Vol. C-25, No. 3, March 1976, pp.228-236.

[PoHK96] Postma, A., Hartman, G., and Krol, Th., Removal of All Faulty Nodes from a Fault-Tolerant Service by Means of Distributed Diagnosis with Imperfect Fault Coverage, in: **Dependable Computing - EDCC-2, Second European Dependable Computing Conference, Taormina, Italy, October 1996, Proceedings**, Hlawiczka, A., Silva, J.G., and Simoncini, L. (Eds.), Springer-Verlag, Berlin, LNCS 1150, 1996, pp.385-402.

[PrMC67] Preparata, F., Metze, G., Chien, R., On the Connection Assignment of Diagnosable Systems, in: **IEEE Transactions on Electronic Computing**, Vol. EC-16, No. 6, Dec. 1967, pp.848-854.

APPENDIX 5.1: Proof of correctness of the reduction rules

In this appendix, we will prove that the reduction rules are correct, i.e., by application of the reduction rules only faulty processors are put in the fault-set F and only correct processors are put in the correct-set C. In other words, we prove, that application of the reduction rules preserves the correctness of:

$$\forall\, n \in F : \mathit{faulty}(n)$$
$$\forall\, n \in C : \mathit{correct}(n)$$

Here, for a processor n, $\mathit{faulty}(n)$ means that n is faulty, whereas $\mathit{correct}(n)$ means that the processor n is correct.

Formally, we can denote the assumptions as follows:

A5.1. $\forall\, a,b \in N :(\mathit{correct}(a) \wedge \mathit{correct}(b)) \Rightarrow M_{a,b} = C$

A5.2. $\forall\, a \in N : \mathit{faulty}(a) \Rightarrow (\exists\, b \in N : \mathit{correct}(b) \wedge (M_{a,b} = F \vee M_{b,a} = F))$

A5.3. $\forall\, a \in N : (\mathit{faulty}(a) \Rightarrow \neg\, \mathit{correct}(a)) \wedge (\neg\, \mathit{correct}(a) \Rightarrow \mathit{faulty}(a))$

A5.4. $\forall\, a,b \in N : (M_{a,b} = F \Rightarrow \neg\,(M_{a,b} = C)) \wedge (\, M_{a,b} = C \Rightarrow \neg\,(M_{a,b} = F))$
A5.5. $|\,\{\, a \in N : faulty(a)\}\,| \le T$

By applying reduction rules to the elements of set N , the number of elements in F respectively C increase. In our proofs, we will use the notations F_i respectively C_j , to indicate the fault-set F containing i processors, respectively the correct-set C containing j processors. Analoguously, we use the notation $P_{i,j}$ to indicate a presupposed correct-set which is element of $PS(n,N - F_i - C_j)$.

By induction on the number of elements i respectively j in F_i and C_j , we prove that the predicates: $(\forall\, n \in F_i : faulty(n))$ and $(\forall\, n \in C_j : correct(n))$ are invariant under application of the reduction rules.

Proof (By induction):
For arbitrary fault-set F_i and correct-set C_j , the reduction rules are as follows:
R5.1. $\forall\, n \in N - F_i - C_j : (M_{n,n} = F \Rightarrow F_{i+1} = F_i \cup \{n\})$
R5.2. $\forall\, n \in N - F_i - C_j : |\,\{\, m \in N - F_i \,|\, M_{m,n} = F \vee M_{n,m} = F \,\}\,| > (T - |\,F_i\,|\,) \Rightarrow$
$F_{i+1} = F_i \cup \{n\}$
R5.3. $\forall\, n \in N - F_i - C_j : (\forall\, m \in N - F_i : M_{m,n} = C \wedge M_{n,m} = C\,) \Rightarrow C_{j+1} = C_j \cup \{n\}$
R5.4. $\forall\, n \in N - F_i - C_j : (\,\neg\,(\exists\, P_{i,j} \in PS(n,N - F_i - C_j) :$
$|\, N - F_i - C_j - P_{i,j}\,| \le T - |\,F_i\,|)) \Rightarrow F_{i+1} = F_i \cup \{n\}$

Basis:
The basis of the proof by induction is the case $i = 0$ and $j = 0$. Since $F_0 = \varnothing$ and $C_0 = \varnothing$, we may conclude that:

$$\forall\, n \in F_0 : faulty(n)$$
$$\forall\, n \in C_0 : correct(n)$$

Induction step:
The induction hypothesis is:

$$\forall\, n \in F_i : faulty(n) \qquad \text{(C5.1)}$$
and
$$\forall\, n \in C_j : correct(n) \qquad \text{(C5.2)}$$

Assuming the induction hypothesis holds, we must prove the correctness of:

$$\forall\, n \in F_{i+1} : faulty(n)$$
and
$$\forall\, n \in C_{j+1} : correct(n)$$

From the definition of the reduction rules, we know that:
$\forall\, n \in N - F_i - C_j : (n \in F_{i+1} - F_i \Rightarrow$
$(M_{n,n} = F \vee$
$|\,\{\, m \in N - F_i \,|\, M_{m,n} = F \vee M_{n,m} = F \,\}\,| > (T - |\,F_i\,|\,) \vee$
$(\,\neg\,(\exists\, P_{i,j} \in PS(n,N - F_i - C_j\,) : |\, N - F_i - C_j - P_{i,j}\,| \le T - |\,F_i\,|)))$ (C5.3)
i.e., a processor n is only added to fault-set F_i , if at least one of the premises of the reduction rules R5.1, R5.2, or R5.4 holds for processor n with respect to F_i.

We will now prove that if one of the premises of the reduction rules R5.1, R5.2, or

R5.4 holds for processor *n* with respect to F_i, then *n* is faulty, i.e.

$$\forall\, n \in N - F_i - C_j : (M_{n,n} = F \vee \; |\{ m \in N - F_i \mid M_{m,n} = F \vee M_{n,m} = F \}| > (T - |F_i|) \vee \; (\neg (\exists\, P_{i,j} \in PS(n, N - F_i - C_j) : |N - F_i - C_j - P_{i,j}| \le T - |F_i|))) \Rightarrow faulty(n) \qquad \text{(C5.4)}$$

From C5.3 and C5.4, we may conclude that:

$$\forall\, n \in N - F_i - C_j : (n \in F_{i+1} - F_i \Rightarrow faulty(n)) \qquad \text{(C5.5)}$$

From C5.1 and C5.5, we may conclude that:

$$\forall\, n \in F_{i+1} : faulty(n) \qquad \text{(C5.6)}$$

Analoguously, from the definition of the reduction rules, we know that:

$$\forall\, n \in N - F_i - C_j : (n \in C_{j+1} - C_j \Rightarrow (\forall\, m \in N - F_i : M_{m,n} = C \wedge M_{n,m} = C)) \qquad \text{(C5.7)}$$

i.e., a processor *n* is only added to correct-set C_j, if the premise of reduction rule R5.3 holds for processor *n* with respect to C_j.

We will now prove that if the premise of reduction rule R5.3 holds for processor *n* with respect to C_j, then *n* is correct, i.e.

$$\forall\, n \in N - F_i - C_j : (\forall\, m \in N - F_i : M_{m,n} = C \wedge M_{n,m} = C) \Rightarrow correct(n) \qquad \text{(C5.8)}$$

From C5.7 and C5.8, we may conclude that:

$$\forall\, n \in N - F_i - C_j : (n \in C_{j+1} - C_j \Rightarrow correct(n)) \qquad \text{(C5.9)}$$

From C5.2 and C5.9, we may conclude that:

$$\forall\, n \in C_{j+1} : correct(n) \qquad \text{(C5.10)}$$

Proof of C5.4 and C5.8:

R5.1. We must prove that:

$\forall\, n \in N - F_i - C_j : (M_{n,n} = F \Rightarrow faulty(n))$

From A5.1 and $N - F_i - C_j \subset N$, we know:

$(\forall\, a,b \in N - F_i - C_j : (correct(a) \wedge correct(b)) \Rightarrow M_{a,b} = C)$

$(\forall\, a,b \in N - F_i - C_j : (correct(a) \wedge correct(b)) \Rightarrow M_{a,b} = C)$

$\Rightarrow$ {Select $a = b$}

$(\forall\, a \in N - F_i - C_j : correct(a) \Rightarrow M_{a,a} = C)$

$\Rightarrow$ {assumption A5.3 and A5.4: $(\neg faulty(a) \Rightarrow correct(a))$ and $(M_{a,a} = C \Rightarrow \neg (M_{a,a} = F)$}

$(\forall\, a \in N - F_i - C_j : \neg faulty(a) \Rightarrow \neg M_{a,a} = F)$

$\Rightarrow$

$(\forall\, a \in N - F_i - C_j : M_{a,a} = F \Rightarrow faulty(a))$

R5.2. We must prove that:

$\forall\, n \in N - F_i - C_j : |\{ m \in N - F_i \mid M_{m,n} = F \vee M_{n,m} = F \}| > (T - |F_i|)$
$\Rightarrow faulty(n)$

$|\{ m \in N - F_i \mid M_{m,n} = F \vee M_{n,m} = F \}| > (T - |F_i|)$

$\Rightarrow$ {assumption A5.3 $\Rightarrow \forall\, n \in N : (faulty(n) \vee correct(n))$ }

$(faulty(n) \vee correct(n)) \wedge$

$(\mid \{ m \in N - F_i \mid M_{m,n} = F \vee M_{n,m} = F \} \mid > (T - \mid F_i \mid))$

$\Rightarrow$ { distributivity }

$(faulty(n) \wedge \mid \{ m \in N - F_i \mid M_{m,n} = F \vee M_{n,m} = F \} \mid > (T - \mid F_i \mid)) \vee$
$(correct(n) \wedge \mid \{ m \in N - F_i \mid M_{m,n} = F \vee M_{n,m} = F \} \mid > (T - \mid F_i \mid))$

$\Rightarrow$ $\{ \mathrm{A} \wedge \mathrm{B} \Rightarrow \mathrm{A}\}$

$(faulty(n)) \vee$
$(correct(n) \wedge \mid \{ m \in N - F_i \mid M_{m,n} = F \vee M_{n,m} = F \} \mid + \mid F_i \mid > T)$

$\Rightarrow$ {assumption A5.1 $\Rightarrow$
$\forall\, m \in N : (correct(n) \wedge (M_{m,n} = F \vee M_{n,m} = F) \Rightarrow faulty(m))$ }

$(faulty(n)) \vee (\mid \{ m \in N - F_i \mid faulty(m) \} \mid + \mid F_i \mid > T)$

$\Rightarrow$ { induction hypothesis $\Rightarrow \forall\, n \in F_i : faulty(n)$ }

$(faulty(n)) \vee (\mid \{ m \in N - F_i \mid faulty(m) \} \mid + \mid \{ m \in F_i \mid faulty(m)\} \mid > T)$

$\Rightarrow$ {assumption A5.5 $\Rightarrow (\mid \{ m \in N \mid faulty(m) \} \mid \leq T)$ }

$faulty(n)$

R5.3: We must prove that:
$\forall\, n \in N - F_i - C_j : (\forall\, m \in N - F_i : M_{m,n} = C \wedge M_{n,m} = C) \Rightarrow correct(n)$

$(\forall\, m \in N - F_i : M_{m,n} = C \wedge M_{n,m} = C)$

$\Rightarrow$

$\neg (\exists\, m \in N - F_i : M_{m,n} = F \vee M_{n,m} = F)$

$\Rightarrow$

$\neg (\exists\, m \in N - F_i : correct(m) \wedge (M_{m,n} = F \vee M_{n,m} = F))$

$\Rightarrow$ {induction hypothesis $\Rightarrow \forall\, n \in F_i : faulty(n)$ }

$\neg (\exists\, m \in N : correct(m) \wedge (M_{m,n} = F \vee M_{n,m} = F))$

$\Rightarrow$ {assumption A5.2 }

$\neg\, faulty(n)$

$\Rightarrow$ {assumption A5.3 }

$correct(n)$

R5.4. We must prove that:
$\forall\, n \in N - F_i - C_j : (\neg (\exists\, P_{i,j} \in PS(n,N - F_i - C_j) :$
$\mid N - F_i - C_j - P_{i,j} \mid \leq T - \mid F_i \mid)) \Rightarrow faulty(n)$

We prove the following equivalent predicate:
$\forall\, n \in N - F_i - C_j : (correct(n) \Rightarrow (\exists\, P_{i,j} \in PS(n,N - F_i - C_j) :$
$\mid N - F_i - C_j - P_{i,j} \mid \leq T - \mid F_i \mid)$

Let $P_{i,j} = \{n \in N - F_i - C_j \mid correct(n)\}$. Clearly, $correct(n) \Rightarrow n \in P_{i,j}$. For this set $P_{i,j}$, we prove that $P_{i,j} \in PS(n,N - F_i - C_j)$ and $\mid N - F_i - C_j - P_{i,j} \mid \leq T - \mid F_i \mid$.

$P_{i,j} \in PS(n,N - F_i - C_j)$

$\Leftarrow$ {definition of $PS(n,N - F_i - C_j)$ }

$n \in P_{i,j} \wedge S(P_{i,j}, N - F_i - C_j) \cap P_{i,j} = \varnothing$

$\Leftarrow$ {definition of $S (P_{i,j}, N - F_i - C_j)$ }

$n \in P_{i,j} \wedge \{m \in N - F_i - C_j \mid i \in P_{i,j} \wedge (M_{m,i} = F \vee M_{i,m} = F)\} \cap P_{i,j} = \varnothing$

$\Leftarrow$ $\{$ $correct(i) \Leftarrow i \in P_{i,j}$ $\}$

$n \in P_{i,j} \wedge \{m \in N - F_i - C_j \mid correct(i) \wedge (M_{m,i} = F \vee M_{i,m} = F)\} \cap$
$\{n \in N - F_i - C_j \mid correct(n)\} = \varnothing$

$\Leftarrow$ {assumption A5.1 $\Rightarrow$
$\forall\, m \in N - F_i - C_j : (correct(i) \wedge (M_{m,i} = F \vee M_{i,m} = F) \Rightarrow faulty(m))$ }

$n \in P_{i,j} \wedge \{m \in N - F_i - C_j \mid faulty(m)\} \cap \{n \in N - F_i - C_j \mid correct(n)\} = \varnothing$

$\Leftarrow$ {assumption A5.3, and $N - F_i - C_j \subset N$ }

$n \in P_{i,j}$

$\Leftarrow$ {definition of $P_{i,j}$ }

$correct(n)$

$| N - F_i - C_j - P_{i,j} | \le T - | F_i |$

$\Leftarrow$ {assumption A5.3, and def. of $P_{i,j} \Rightarrow \forall\, m \in N - F_i - C_j - P_{i,j} : faulty(m)$ }

$| \{m \in N - F_i - C_j - P_{i,j} \mid faulty(m)\} | \le T - | F_i |$

$\Leftarrow$ {induction hypothesis $\Rightarrow \forall\, n \in F_i : faulty(n)$ }

$| \{m \in N - F_i - C_j - P_{i,j} \mid faulty(m)\} | \le T - | \{m \in F_i \mid faulty(m)\} |$

$\Leftarrow$

$| \{m \in N - C_j - P_{i,j} \mid faulty(m)\} | \le T$

$\Leftarrow$ $\{$ $correct(i) \Leftarrow i \in P_{i,j}$ and assumption A5.3 }

$| \{m \in N - C_j \mid faulty(m)\} | \le T$

$\Leftarrow$ {induction hypothesis $\Rightarrow \forall\, n \in C_j : correct(n)$ and assumption A5.3}

$| \{m \in N \mid faulty(m)\} | \le T$

$\Leftarrow$ {assumption A5.5 }

TRUE ❑

APPENDIX 5.2: Proof of consistency of the reduction rules

In this appendix, we will prove the consistency of the reduction rules as formulated below:

1. $\forall\, n \in N - F_i - C_j : (M_{n,n} = F \Rightarrow F_{i+1} = F_i \cup \{n\})$
2. $\forall\, n \in N - F_i - C_j : | \{ m \in N - F_i \mid M_{m,n} = F \vee M_{n,m} = F \} | > (T - | F_i |) \Rightarrow$
   $F_{i+1} = F_i \cup \{n\}$
3. $\forall\, n \in N - F_i - C_j : (\forall\, m \in N - F_i : M_{m,n} = C \wedge M_{n,m} = C) \Rightarrow C_{j+1} = C_j \cup \{n\}$
4. $\forall\, n \in N - F_i - C_j : (\neg (\exists\, P_{i,j} \in PS(n, N - F_i - C_j) :$
   $| N - F_i - C_j - P_{i,j} | \le T - | F_i |) \Rightarrow F_{i+1} = F_i \cup \{n\}$

First, reduction rule 1 is applied to all processors for which $M_{n,n} = F$. Then, reduction rules 2 through 4 may be applied in any order. In this appendix, we will prove that:

C5.1. $\forall a \in \{1,2,3,4\}$: $(\exists\, x \in N - F_i - C_j$: reduction rule a can be applied to processor x for fault-set F_i) $\wedge F_{i+1} = F_i \cup \{y\} \wedge (y \ne x) \Rightarrow$
(reduction rule a can be applied to processor x for fault-set F_{i+1})

i.e., putting a processor y in the fault-set does not change the applicability of the reduction rules 1 through 4 for processor x.

C5.2. $\forall a \in \{1,2,3,4\}$: ($\exists\, x \in N - F_i - C_j$: reduction rule a can be applied to processor x for correct-set C_j) $\wedge\ C_{j+1} = C_j \cup \{y\} \wedge (y \neq x) \Rightarrow$
(reduction rule a can be applied to processor x for correct-set C_{j+1})

i.e., putting a processor y in the correct-set does not change the applicability of the reduction rules 1 through 4 for processor x.

Furthermore, we must prove that the reduction rules 2 through 4 are consistent, i.e., it must be impossible to classify a processor both in the fault-set and in the correct-set. A processor x is put in the correct-set only if the premise of reduction rule 3 is satisfied. We prove that if a processor y satifies the premise of reduction rule 2 or 4, processor x and y must be two different processors. Thus:

C5.3. $\forall\, a \in \{2,4\}$:
($\exists\, x \in N - F_i - C_j$: reduction rule a can be applied to processor x) $\wedge$
($\exists\, y \in N - F_i - C_j$: reduction rule 3 can be applied to processor y) $\Rightarrow (y \neq x)$

Proof of C5.1:
***a* = 1**
$\exists\, x \in N - F_i - C_j : (M_{x,x} = F) \wedge F_{i+1} = F_i \cup \{y\} \wedge (y \neq x) \Rightarrow (M_{x,x} = F)$

***a* = 2**
$\exists\, x \in N - F_i - C_j : |\{ k \in N - F_i \mid M_{k,x} = F \vee M_{x,k} = F \}| > (T - |F_i|) \wedge$
$F_{i+1} = F_i \cup \{y\} \wedge (y \neq x) \Rightarrow$
$|\{ k \in N - F_{i+1} \mid M_{k,x} = F \vee M_{x,k} = F \}| + 1 \geq$
$|\{ k \in N - F_i \mid M_{k,x} = F \vee M_{x,k} = F \}| >$
$(T - |F_i|) \geq (T - |F_{i+1}| + 1) \Rightarrow$
$|\{ k \in N - F_{i+1}{}' \mid M_{k,x} = F \vee M_{x,k} = F \}| \geq (T - |F_{i+1}|)$

***a* = 3**
$\exists\, x \in N - F_i - C_j : (\forall\, m \in N - F_i : M_{m,x} = C \wedge M_{x,m} = C) \wedge F_{i+1} = F_i \cup \{y\} \wedge (y \neq x) \Rightarrow$
$(\forall\, m \in N - F_{i+1} : M_{m,x} = C \wedge M_{x,m} = C)$ since $N - F_{i+1} \subset N - F_i$

***a* = 4**
$\exists\, x \in N - F_i - C_j : (\neg (\exists\, P_{i,j} \in PS(x, N - F_i - C_j) : |N - F_i - C_j - P_{i,j}| \leq T - |F_i|) \wedge$
$F_{i+1} = F_i \cup \{y\} \wedge (y \neq x) \Rightarrow$
$(\neg (\exists\, P_{i,j} \in PS(x, N - F_{i+1} - C_j) : |N - F_{i+1} - C_j - P_{i,j}| \leq T - |F_{i+1}|)$
since $PS(x, N - F_{i+1} - C_j) \subset PS(x, N - F_i - C_j) \wedge |N - F_{i+1} - C_j - P_{i,j}| \leq T - |F_{i+1}|$
The predicate $|N - F_{i+1} - C_j - P_{i,j}| \leq T - |F_{i+1}|$ is implied by:
$|N - F_{i+1} - C_j - P_{i,j}| + 1 = |N - F_i - C_j - P_{i,j}|$ and
$|N - F_i - C_j - P_{i,j}| \leq T - |F_i|$ and
$T - |F_i| = T - |F_{i+1}| + 1$

Proof of C5.2:
***a* = 1**
$\exists\, x \in N - F_i - C_j : (M_{x,x} = F) \wedge C_{j+1} = C_j \cup \{y\} \wedge (y \neq x) \Rightarrow (M_{x,x} = F)$

***a* = 2**
$\exists\, x \in N - F_i - C_j : |\, \{ k \in N - F_i \mid M_{k,x} = F \vee M_{x,k} = F \}\, | > (T - |\, F_i\,|\,) \wedge$
$C_{j+1} = C_j \cup \{y\} \wedge (y \neq x) \Rightarrow$
$|\, \{ k \in N - F_i \mid M_{k,x} = F \vee M_{x,k} = F \}\, | \geq (T - |\, F_i\,|\,)$

***a* = 3**
$\exists\, x \in N - F_i - C_j : ((\forall\, m \in N - F_i : M_{m,x} = C \wedge M_{x,m} = C) \wedge C_{j+1} = C_j \cup \{y\} \wedge (y \neq x)$
$\Rightarrow$
$(\forall\, m \in N - F_i : M_{m,x} = C \wedge M_{x,m} = C)$

***a* = 4**
We must prove that:
$\exists\, x \in N - F_i - C_j : (\, \neg\, (\exists\, P_{i,j} \in PS(x, N - F_i - C_j) : |\, N - F_i - C_j - P_{i,j}\,| \leq T - |\, F_i\,|) \wedge$
$C_{j+1} = C_j \cup \{y\} \wedge (y \neq x) \Rightarrow$
$(\, \neg\, (\exists\, P_{i,j} \in PS(x, N - F_i - C_{j+1}) : |\, N - F_i - C_{j+1} - P_{i,j}\,| \leq T - |\, F_i\,|)$

Proof:
We distinguish two cases: $y \in P_{i,j}$ and $y \notin P_{i,j}$
Case $y \in P_{i,j}$
$\exists\, x \in N - F_i - C_j : (\forall\, P_{i,j} \in PS(x, N - F_i - C_j) : |\, N - F_i - C_j - P_{i,j}\,| > T - |\, F_i\,|) \wedge$
$C_{j+1} = C_j \cup \{y\} \wedge (y \neq x) \wedge (y \in P_{i,j})$
$\Rightarrow$ $\{\, C_{j+1} = C_j \cup \{y\} \wedge (y \in P_{i,j}) \Rightarrow |\, N - F_i - C_j - P_{i,j}\,| = |\, N - F_i - C_{j+1} - P_{i,j}\,|\}$
$(\forall\, P_{i,j} \in PS(x, N - F_i - C_j) : |\, N - F_i - C_{j+1} - P_{i,j}\,| > T - |\, F_i\,|)$
$\Rightarrow \{\, PS(x, N - F_i - C_{j+1}) \subset PS(x, N - F_i - C_j)\, \}$
$(\forall\, P_{i,j} \in PS(x, N - F_i - C_{j+1}) : |\, N - F_i - C_{j+1} - P_{i,j}\,| > T - |\, F_i\,|)$

Case $y \notin P_{i,j}$
$\exists\, x \in N - F_i - C_j : (\forall\, P_{i,j} \in PS(x, N - F_i - C_j) : |\, N - F_i - C_j - P_{i,j}\,| > T - |\, F_i\,|) \wedge$
$C_{j+1} = C_j \cup \{y\} \wedge (y \neq x) \wedge (y \notin P_{i,j})$
$\Rightarrow \{$ R5.3 $\Rightarrow (y \in C_j \Rightarrow (\forall\, m \in N - F_i : M_{m,y} = C \wedge M_{y,m} = C),$
definition $PS(x, N - F_i - C_j)\}$
$(\forall\, P_{i,j} \in PS(x, N - F_i - C_j)$: $P_{i,j+1} = P_{i,j} \cup \{y\} \Rightarrow P_{i,j+1} \in PS(x, N - F_i - C_j) \wedge$
$(\forall\, P_{i,j+1} \in PS(x, N - F_i - C_j) : |\, N - F_i - C_j - P_{i,j+1}\,| > T - |\, F_i\,|)$
$\Rightarrow \{((P_{i,j+1} = P_{i,j} \cup \{y\}) \wedge (C_{j+1} = C_j \cup \{y\}\,)) \Rightarrow$
$|\, N - F_i - C_j - P_{i,j+1}\,| = |\, N - F_i - C_{j+1} - P_{i,j}\,|\,\}$
$(\forall\, P_{i,j} \in PS(x, N - F_i - C_j) : |\, N - F_i - C_{j+1} - P_{i,j}\,| > T - |\, F_i\,|)$

Proof of C5.3:
***a* = 2**
$\exists\, x \in N - F_i - C_j : |\, \{ k \in N - F_i \mid M_{k,x} = F \vee M_{x,k} = F \}\, | > (T - |\, F_i\,|\,) \wedge$
$\exists\, y \in N - F_i - C_j : (\forall\, m \in N - F_i : M_{m,y} = C \wedge M_{y,m} = C)$
$\Rightarrow$ { assumption A5.4 $\Rightarrow ((\forall\, a, b \in N: M_{a,b} = C\,) \Rightarrow \neg M_{a,b} = F)\}$
$|\, \{ k \in N - F_i \mid M_{k,x} = F \vee M_{x,k} = F \}\, | > (T - |\, F_i\,|\,) \wedge$
$|\, \{ k \in N - F_i \mid M_{k,y} = F \vee M_{y,k} = F \}\, | = 0$
$\Rightarrow$ { Appendix 1 $\Rightarrow (\forall\, f \in F_i : faulty(f) \Rightarrow F_i = \{\, f \in F_i \mid faulty(f)\}\,)\,\}$

$| \{ k \in N - F_i \mid M_{k,x} = F \vee M_{x,k} = F \} | > (T - | \{ f \in F_i \mid faulty(f) \} |) \wedge$
$| \{ k \in N - F_i \mid M_{k,y} = F \vee M_{y,k} = F \} | = 0$

$\Rightarrow$ {assumption A5.5 $\Rightarrow$
$(| \{ f \in F_i \mid faulty(f) \} | \leq T) \Rightarrow (T - | \{ f \in F_i \mid faulty(f) \} | \geq 0) \}$
$| \{ k \in N - F_i \mid M_{k,x} = F \vee M_{x,k} = F \} | > 0 \wedge$
$| \{ k \in N - F_i \mid M_{k,y} = F \vee M_{y,k} = F \} | = 0$

$\Rightarrow$
$(y \neq x)$

***a* = 4**

$\exists\, x \in N - F_i - C_j : (\neg (\exists\, P_{i,j} \in PS(x, N - F_i - C_j) : | N - F_i - C_j - P_{i,j} | \leq T - | F_i |)) \wedge$
$\exists\, y \in N - F_i - C_j : (\forall\, m \in N - F_i : M_{m,y} = C \wedge M_{y,m} = C)$

$\Rightarrow$ { Appendix 1, proof R5.3 $\Rightarrow$
$(\forall\, n \in N - F_i - C_j : (\forall\, m \in N - F_i : M_{m,n} = C \wedge M_{n,m} = C) \Rightarrow correct(n)) \}$
$\exists\, x \in N - F_i - C_j :(\neg (\exists\, P_{i,j} \in PS(x, N - F_i - C_j) : | N - F_i - C_j - P_{i,j} | \leq T - | F_i |)) \wedge$
$(y \in N - F_i - C_j) \wedge correct(y)$

$\Rightarrow$ { Appendix 1, proof R5.4: $\forall\, n \in N : correct(n) \Rightarrow$
$(\exists\, P_{i,j} \in PS(n, N - F_i - C_j) :| N - F_i - C_j - P_{i,j} | \leq T - | F_i |) \}$
$\exists\, x \in N - F_i - C_j : (\neg (\exists\, P_{i,j} \in PS(x , N - F_i - C_j) : | N - F_i - C_j - P_{i,j} | \leq T - | F_i |) \wedge$
$\exists\, y \in N - F_i - C_j : ((\exists\, P_{i,j} \in PS(y , N - F_i - C_j) : | N - F_i - C_j - P_{i,j} | \leq T - | F_i |)$

$\Rightarrow$
$(y \neq x)$

CHAPTER 6

Conclusion

MACBETH:	*"What news more ?*
SEYTON:	*All is confirmed, my lord, which was reported."*

William Shakespeare, Macbeth

Abstract

In this chapter, an evaluation is given of the results obtained in the previous chapters. This chapter concludes with some suggestions for further research.

6.1. Evaluation of results

This thesis has concentrated on describing several Byzantine fault-tolerant protocols for a system based on hardware masking redundancy techniques. The protocols are designed to work in a distributed system consisting of a set, *N*, of *N* processors, and consist of a set of replicated processes, each replica executed on a different processor. The protocols are fault-tolerant in such a way that an arbitrary set of up to *T* processors in the system is allowed to behave maliciously. Here, *T* is a design parameter, and, at the same time, a measure for the reliability of the system, which is independent of the technical quality of the components of the system under consideration. Implicitly, the protocols are based on the assumption that the higher the value of *T*, the more reliable the system will be. In other words, it is assumed that processors fail independently. This assumption is justified by the fact that our protocols are designed to work in a distributed system, since the autonomous nature and geographical distribution of the processors in a distributed system contribute significantly to achieving independence between the failures of different processors in the system.

We will now give a short summary of the protocols introduced in this thesis. Section 6.1.1. summarizes the new authenticated Byzantine Agreement Protocols described in Chapter 3 of this thesis. Section 6.1.2. recalls the distributed cryptographic function application protocols (DCFAPs) investigated in Chapter 4. Finally, Section 6.1.3. looks back on the Byzantine fault-tolerant distributed diagnosis algorithm introduced in Chapter 5.

As an introduction to the Byzantine fault-tolerant protocols introduced in this thesis and summarized below, in Chapter 2, a description of the so-called Dependable Distributed Data Storage (DDDS) system has been given. The DDDS system is a virtual system aimed at providing reliable distributed data storage, which consists of a set, *N*, of *N* processors, an arbitrary set of up to *T* of which is allowed to behave maliciously.

In any implementation of the DDDS system, the protocols summarized below can conveniently be applied.

6.1.1. Authenticated Byzantine Agreement Protocols

In Chapter 3, several extensions have been proposed to existing solutions of the well-known Byzantine Generals Problem, called Byzantine Agreement Protocols (or BAPs). The first BAPs were proposed by Lamport et al. in [LaSP82] for a lock-step synchronous system, in which all correct processors know the start of the BAP and start the BAP simultaneously.

BAPs are designed to work in a replicated system (i.e., a system based on system-level hardware masking redundancy techniques) and aim to avoid inconsistency between the correct replicas due to broadcast errors caused by a malfunctioning source processor providing input data to the replicas in the system. It is important to avoid such inconsistencies, since they may lead to a system breakdown, even if the system does not contain more faulty processors than it is designed to tolerate. Because any replicated fault-tolerant system is basically connected to an unreliable source processor (i.e., a processor providing input data to the system) and hence, has to cope with possible broadcast errors, BAPs are an invaluable ingredient for any system in which arbitrary processor faults should be tolerated.

A disadvantage of the BAPs proposed in [LaSP82] is that they require much communication overhead and a large number of processors. In [LaSP82], Lamport et al. distinguish between authenticated and non-authenticated BAPs. In contrast to non-authenticated BAPs, authenticated BAPs require the use of message authentication, for which, since a Byzantine fault model is assumed, application of cryptographic techniques is indispensable. A disadvantage of applying cryptographic techniques for message authentication is, that the correctness of the BAP becomes dependent of the security of the cryptosystem that is used, and application of cryptographic techniques increases the computation effort needed to generate and verify messages created during execution of the BAP. Arguments in favour of the use of message authentication are that it drastically reduces the required amount of message communication, as well as the number of processors that is needed for the BAP to work.

However, even for the *authenticated* BAPs originally proposed by Lamport et al. in [LaSP82], the required communication overhead is often prohibitively large. Therefore, many attempts have been made in order to reduce the required amount of message communication in authenticated BAPs. Reduction of the amount of message communication required in an authenticated BAP can be obtained in several ways, a number of which have been described in literature. In Section 3.2. of this thesis, we have investigated a new class of authenticated BAPs in which the reduction of the communication overhead is obtained in an alternative way, viz., by reduction of the size of the messages.

The large communication overhead required in most BAPs is not the only reason for their restricted applicability, however. A more serious disadvantage of the BAPs proposed by Lamport et al. in [LaSP82], and also of many other BAPs proposed in literature, is that these BAPs are only applicable in a lock-step synchronous system, in which all correct processors know the start of the BAP and start the BAP simultaneously. In a distributed system, the correct processors usually do not have a common notion of time, and as a consequence, it is very hard to have all correct processors start the BAP simultaneously. This makes many BAPs described in literature inconvenient for use in a distributed system.

To make authenticated BAPs also applicable in a distributed system, in [CASD95], a new class of BAPs has been proposed, which does not require the correct processors to know the start of the BAP or to start the BAP simultaneously. However, these BAPs require the processor clocks of all correct processors to be approximately synchronized at the start and during execution of the BAP, which requires the system to contain at least a majority of correctly functioning processors, which periodically execute a fault-tolerant clock synchronization algorithm, in order to be able to guarantee that the clock skew between the processor clocks of correct processors remains bounded.

In this thesis, we have observed that in fact, for an authenticated BAP to work, it is sufficient that during execution of the BAP, approximate *protocol synchronization* (i.e., approximately simultaneous execution of the protocol on all correct processors) exists between the correct processors. Approximate clock synchronization between the processor clocks of correct processors, as required in the BAPs in [CASD95], is only a means to establish approximate protocol synchronization between the correct processors during execution of the BAP. However, the price that has to be paid to establish approximately synchronized clocks is a required majority of correct processors in the system and required periodical execution of a fault-tolerant clock synchronization algorithm. As is shown in this thesis, approximate protocol synchronization between the correct processors can also simply be achieved *during execution of the BAP*, *without* requiring the clocks of correct processors to be approximately synchronized at the start or during execution of the BAP.

To show this, in Section 3.3., we have introduced a new class of so-called authenticated self-synchronizing BAPs, which guarantee Byzantine Agreement for any number of faulty processors in the system and arbitrary clock skew between the clocks of the processors in the system. In these authenticated self-synchronizing BAPs, the processors need not know the start of the protocol. Approximate protocol synchronization between the correct processors in the system is established during execution of the BAP.

A disadvantage of these authenticated self-synchronizing BAPs is that they require a considerable communication overhead, part of which is used to establish approximate protocol synchronization between the correct processors in the system. Fortunately, in these BAPs, a reduction of the required amount of message communication can be established by reducing the number of destination processors to which a message is sent. Application of this message communication reduction strategy in the authenticated self-synchronizing BAPs has led to the so-called optimized authenticated self-synchronizing BAPs, which require less communication overhead than the original authenticated self-synchronizing BAPs (for $N > T+2$), be it at the cost of increased lengths of the communication phases of the BAP (for $T > 1$), if compared to the original authenticated self-synchronizing BAP.

Finally, in Section 3.4., solutions applicable for a distributed system have been given for a problem that is closely related to the Byzantine Generals Problem, i.e. the problem of guaranteeing agreement between all correct processors in a system of *N* processors about *N* initial values, each of which is held by a different processor. Up to *T* processors in the system are allowed to behave maliciously. Solutions to this problem, called interactive consistency algorithms (or ICAs), were originally proposed by Pease et al. in [PeSL80] for a lock-step synchronous system in which all correct processors know the start of the algorithm and start the algorithm simultaneously.

As stated before, in a distributed system, the correct processors usually do not have a common notion of time, and hence it is very hard to have all correct processors start the algorithm simultaneously. This makes the ICA described in [PeSL80] inconvenient for use in a distributed system.

In a lock-step synchronous system the solution boils down to simultaneous execution of *N* BAPs, one for each initial value held by a different processor. To make an ICA applicable also in a distributed system, approximate protocol synchronization between the different BAPs that are part of the ICA should be established. Based on the self-synchronizing BAPs introduced in Section 3.3., we have presented two different types of self-synchronizing ICAs: authenticated self-synchronizing ICAs (based on authenticated self-synchronizing BAPs) and optimized authenticated self-synchronizing ICAs (based on optimized authenticated self-synchronizing BAPs), respectively.

6.1.2. Distributed cryptographic function application protocols

Chapter 4 has been devoted to so-called distributed cryptographic function application protocols (DCFAPs). These protocols are designed to work in a system consisting of a set, *N*, of *N* processors, up to *T* of which are allowed to behave maliciously. Provided that the number of faulty processors in the system is less than or equal to *T*, DCFAPs establish encryption of a certain commonly known piece of data, *B*, with a secret cryptographic function *F*, and guarantee agreement between the correct processors in the system about the result of encryption of data *B* with function *F*. Each processor in the system is given the capability to perform only part of the encryption with function *F*. A sufficient number of processors should cooperate in order to be able to generate or apply *F*. The capabilities are chosen such, that it is computationally infeasible for a set of up to *T* colluding processors to generate or apply *F* without the help of one or more correct processors.

Two different types of DCFAPs have been introduced: interactive DCFAPs based on a fault-tolerant version of the multisignature scheme proposed by Okamoto in [Okam88], and non-interactive DCFAPs based on the principle of function sharing proposed by De Santis et al. in [DDFY94]. In most cases, the non-interactive DCFAPs are preferred to the interactive DCFAPs, since, for $T > 1$, the minimally required number of processors is smaller than in the interactive DCFAPs, and, moreover, for $T > 0$, the non-interactive DCFAPs require less computation overhead than the interactive DCFAPs. For $T = 0$ or $T = 1$, the minimally required number of processors is equal for both types of DCFAPs, whereas, for $T = 0$, it also holds that the required communication overhead is equal for both types of DCFAPs.

In Chapter 4, we have shown that DCFAPs can conveniently be used in a recovery process responsible for recovery of lost or corrupted data fragments in a dependable distributed data storage system in which data is stored in a fault-tolerant and secure way on a set, N, of N processors, up to T of which are allowed to behave maliciously. The explanation for this can be summarized as follows.

A failure of a processor in N may lead to loss or corruption of data stored on it. In general, in order to protect data from unauthorized reading and undetected mutilation, any piece of data stored in the system has been encrypted and signed by its owner. Any piece of data that has got lost or corrupted should be regenerated, after which it should be signed. Signing a re-created recovered data fragment requires encryption of its hash value with the secret cryptographic function with which the owner of the data fragment had signed the original data fragment. Since, in general the secret cryptographic function used to sign a piece of data is only known to the owner of that piece of data, it looks straightforward to let every user in the system be responsible for recovery of its own lost or corrupted data fragments. However, this would imply that the capability of recovering a lost or corrupted data fragment would depend on the correctness of its owner's processor. Since the system should be resilient to simultaneous failure of up to T processors, the responsibility for recovery of lost or corrupted data fragments can never be given to one processor, but should be given to a group of processors instead. This group of processors can conveniently use a DCFAP to sign any re-created recovered data fragment with the secret cryptographic function with which the owner had signed the original data fragment that has got lost or corrupted.

6.1.3. A Byzantine fault-tolerant distributed diagnosis algorithm

In Chapter 5, a new Byzantine fault-tolerant distributed diagnosis algorithm has been described. This algorithm has been designed for a system consisting of a set, N, of N processors, up to T of which may behave maliciously. The algorithm is independently executed on every processor. Execution of the diagnosis algorithm yields a set S of processors which includes all detectably faulty processors in the system. Hence, the remaining set of processors $N \setminus S$ is guaranteed to contain only correctly functioning processors. To be able to guarantee that a fault-tolerant system continues to satisfy its specification despite the presence of maliciously functioning processors, it is of key importance that all detectably faulty processors are removed from the system as soon as possible (and replaced by correctly functioning processors). So, after having determined the set S of detectably faulty processors, the processors in S should be removed from the system as soon as possible.

It is assumed that, prior to execution of the diagnosis algorithm, every processor has tested every other processor in the system, and that the results of these tests are available in every correct processor. It is also assumed that all correct processors agree on these test results. BAPs can be applied in order to guarantee agreement among the correct processors about the test results.

We do not further specify what tests are performed, however, we allow the tests to have only an imperfect fault coverage. Imperfect fault coverage means that a correct processor p which tests a faulty processor q will not always detect that processor q is faulty, and hence, in such a case, the outcome of its test will wrongly indicate that processor q is correct. On the contrary, if a correct processor p tests another correct processor q, p will always conclude that q is correct. In practice, tests always have an imperfect fault coverage.

A lot of diagnosis algorithms have appeared in literature. Many diagnosis algorithms are capable of determining the set of faulty processors under the assumption that tests have a 100% fault coverage. This assumption limits the applicability of these algorithms, since, in practice, tests always have an imperfect fault coverage. Other diagnosis algorithms use a probabilistic approach to cope with tests with imperfect fault coverage. Although these algorithms allow tests to have an imperfect fault coverage, they can only guarantee to find the set of processors that are most likely to be faulty.

The new diagnosis algorithm introduced in Chapter 5 combines the best of both worlds by assuring that all detectably faulty processors are removed from the system, while at the same time allowing tests to have an imperfect fault coverage.

6.2. Suggestions for further research

A thesis is never finished. Although in this thesis, solutions have been presented to a number of important problems in the field of fault-tolerant computing, many others have remained unsolved or have only received minor attention. This section attempts to give a few suggestions for future research.

First of all, the DDDS system introduced in Chapter 2 deserves more attention. In this thesis, the system has only been used as an introduction to the protocols presented in later chapters. The DDDS system, which can be built from off-the-shelf hardware components, can be implemented on a distributed system consisting of a set, *N*, of *N* processors, which are connected in a network and up to *T* of which may behave maliciously. It is based on masking redundancy techniques combined with diagnosis and recovery algorithms, and aims at providing a means for several users to store their files in a reliable way. The DDDS system can continue to provide this functionality, as long as the number of maliciously behaving processors in the system remains less than or equal to *T*. In order to avoid single points of failure in the DDDS system, the functionality of the DDDS system is provided by a collection of replicated system services, while, for any of the system services, each replica resides on a different processor. In order to avoid inconsistency between the replicas of a system service, message communication is required between the processors on which the replicas reside. This message communication decreases the performance of the system (viz., as an effect of the required message communication, the time needed to execute the system service increases) and would not be required if the system service would have been implemented on a single system.

Thus, in the DDDS system, an increase in reliability is obtained at the cost of a decrease in system performance, if compared to a single system. Unless great value is attached to the reliability of data storage, today's designers of a data storage system using off-the-shelf hardware components will in general decide in favour of a single system, instead of a DDDS system. However, the rapid improvements that are made in computer technology with regard to the processing speed and the speed of network communication may soon cause this situation to change in favour of the DDDS system.

In Chapter 2, for the DDDS system, a number of system services have been distinguished. Of these system services, the data storage and retrieval service, the data recovery service, and the diagnosis service have been investigated in some depth. On the other hand, the membership management service and the version management service have barely received attention. More research is needed to also investigate the latter two system services.

Furthermore, implementation of (parts of) the DDDS system is strongly suggested, since it may greatly contribute to the insight in practical problems related to the DDDS system. Moreover, it might create an opportunity to obtain quantitative results, e.g., a quantification of the loss in system performance as a function of increasing maximum number of faulty processors, T, that is tolerated.

From the above, it will be clear that much research remains to be done to the DDDS system. Below, we will also give some suggestions for future research with regard to the protocols described in Chapter 3, 4, and 5.

In Figure 3.1. in Section 3.1.4., a classification of BAPs has been given. From this classification, it is apparent that several types of BAPs have not yet appeared in literature, e.g., partially synchronous BAPs or deterministic non-authenticated BAPs for which the processor clocks need not exactly be synchronized, or for which the start of the BAP needs not exactly be synchronized. An interesting direction for future research would be to investigate whether these BAPs do actually exist or not. If these BAPs really exist, the challenge to design them remains.

In Chapter 4, we were faced with the problem of secure generation and distribution of cryptographic keys (Section 4.2.) respectively of share functions of a secret cryptographic function (Section 4.3.). In essence, the problem boils down to finding a way to correctly communicate information from one processor to one or more other processors in a system despite the presence of faulty processors. In cryptography, this problem is commonly referred to as finding a way to establish secure channels between pairs of processors. Actually, the authenticated BAPs presented in Chapter 3 require a solution to the same problem. In these authenticated BAPs, it is assumed that prior to execution of the BAP, all correct processors already have obtained some knowledge. E.g., they should know the identity of all other processors that participate in the BAP, and, also, the public cryptographic function of any other processor participating in the BAP should be a priori known. This is only guaranteed if there exists a way for a correct processor to distribute this information to the other processors in the system, without being hindered by faulty processors in the system. More research is needed to find a solution to this problem.

6.3. References

[CASD95] Cristian, F., Aghili, H., Strong, R., and Dolev, D., Atomic broadcast: From simple message diffusion to Byzantine agreement, in: **Information and Computation**, Vol. 118, 1995, pp. 158-179. (An earlier version of this paper appeared in: **Proceedings of the 15th International Symposium on Fault-Tolerant Computing**, Ann Arbor, June 1985, pp.200-206).

[DDFY94] De Santis, A., Desmedt, Y., Frankel, Y., and Yung, M., How to Share a Function Securely, in: **Proceedings of the 26th Annual ACM Symposium on the Theory of Computing**, Montréal, Canada, 1994, pp.522-533.

[LaSP82] Lamport, L., Shostak, R., and Pease, M., The Byzantine Generals Problem, in: **ACM Transactions on Programming Languages and Systems**, Vol.4, No.3, July 1982, pp.382-401.

[Okam88] Okamoto, T., A Digital Multisignature Scheme Using Bijective Public-Key Cryptosystems, in: **ACM Transactions on Computer Systems**, Vol.6, No.8, November 1988, pp.432-441.

[PeSL80] Pease, M., Shostak, R., and Lamport, L., Reaching agreement in the presence of faults, in: **Journal of the ACM**, Vol.27, No.2, April 1980, pp.228-234.

Abstract

The ever increasing use of computers in almost every part of today's life makes us more and more dependent on the correct functioning of computer systems. Although improvements on the technical quality of the applied hardware components are continuously being made, failures of computer systems can never be ruled out completely. However, the probability of a system failure can be highly reduced by applying techniques of fault-avoidance, fault detection and / or fault-tolerance. Application of each of these techniques is indispensable in critical applications that require a high degree of reliability, i.e., in systems that require the probability of a system failure to be kept as low as possible.

Whereas a system based on fault-avoidance and / or fault detection fails to satisfy its specification as soon as one or more of its components fail, a fault-tolerant system may continue to satisfy its specification despite the failure of one or more of its components. For this purpose, any fault-tolerant system contains some form of redundancy, i.e., the system contains components (hardware redundancy) and / or performs computations (software redundancy) that are superfluous as long as no faults occur. Whereas software redundancy may provide resilience to transient and intermittent faults, hardware redundancy is needed in any system that must be resilient to permanent faults. Hardware redundancy can be applied at different levels in the system, e.g., at gate level or at system level.

This thesis concentrates on the design of new algorithms for fault-tolerant systems based on system-level hardware masking redundancy. It is argued that any system in which a reliability improvement of at least a factor 100 is required should be based on system-level hardware masking redundancy. The technique of system-level hardware masking redundancy is applicable in a redundant system consisting of a number of processors, in which the system services are replicated on the different processors, and provides resilience to a limited number of faulty processors in the system. The technique is most effective in a distributed system, since the autonomous nature and geographical distribution of the processors in such a system largely contribute to achieve independency between failures of different processors, which improves the reliability of the system.

A distributed system consists of a number of processors interconnected via a network. Each processor has its own memory and its own processor clock. Processors in a distributed system can only exchange information by sending messages to each other.

A distributed system can be modeled as a synchronous system. In this thesis, a synchronous system is defined as a system with synchronous communication (i.e., it is assumed that there exists a real-time nonnegative upper bound τ_{max} on the time needed to communicate a message from a correct processor to another processor in the system) and synchronous processors (i.e., the rate at which the processor clock of any cor-

rect processor drifts from real-time, is bounded by a factor $(1+\rho)$, for some $\rho \geq 0$). It is assumed that there also exists a real-time nonnegative lower bound τ_{min} (with $\tau_{min} \leq \tau_{max}$) on the time needed to communicate a message from a correct processor to another processor in the system, and that the bounds τ_{min}, τ_{max}, and $(1+\rho)$ are known by each correct processor in the system. In practice, in a distributed system, there always exists some uncertainty in the time needed to deliver a message (i.e., $\tau_{min} \neq \tau_{max}$), and no two processor clocks run at exactly the same rate (i.e., $\rho > 0$).

In a distributed system, processors are autonomous in that sense that no shared memory and hence, no a priori global knowledge exists between the processors in the system. The processors can only exchange information by sending messages to each other. In particular, in a distributed system, it is impossible to have all correct processors start any distributed algorithm simultaneously, for it is impossible to have all correct processors agree on a common point in time, since every processor has its own processor clock, no two processor clocks run at the same rate, and, due to uncertain message delivery times, any fault-tolerant clock synchronization algorithm can only establish approximate synchronization between the correct processors in the system.

Many fault-tolerant algorithms found in literature require a lock-step synchronous system. A lock-step synchronous system is a synchronous system in which $\tau_{min} = \tau_{max}$, and $\rho = 0$. Furthermore, they require all correct processors to know the start of the algorithm and to start the algorithm simultaneously. Since, in a distributed system, none of the above requirements can be fulfilled, these algorithms are not applicable in a distributed system.

Nearly all algorithms[1] described in this thesis are applicable in any synchronous system (i.e., a synchronous system with arbitrary known nonnegative ρ, τ_{min}, and τ_{max}, with $\tau_{min} \leq \tau_{max}$) consisting of a set, $\mathcal{N}$, of N processors in which the system services are replicated on the various processors in the system. The algorithms are self-synchronizing, i.e., they do not require the correct processors in the system to know the start of the algorithm and to start the algorithm simultaneously. The algorithms satisfy their specification as long as the number of faulty processors in the system is less than or equal to some predefined system parameter, T.

In general, processors may fail in an arbitrary way. Unfortunately, in practice, it is impossible to make the system resilient to arbitrary failures of its processors. To evade this problem, the processors are assumed to fail according to some predefined fault model. The weaker the fault model (i.e., the more general the fault behaviour that is allowed by the fault model), the higher the reliability of the system. In this thesis, any faulty processor in the system is assumed to exhibit a behaviour which is consistent with a certain restricted Byzantine fault model. The proposed fault model is a very weak fault model, which takes into account almost all faults that are relevant for the algorithms under consideration.

1. Only the algorithms described in Section 3.2. require a lock-step synchronous system, in which all correct processors know the start of the algorithm and start the algorithm simultaneously.

This thesis is composed as follows.

In Chapter 1, an introduction to dependable computing is given.

Then, in Chapter 2, the so-called Dependable Distributed Data Storage (DDDS) system is presented. This is a virtual system for reliable data storage, in which the algorithms described in this thesis can conveniently be applied. The DDDS system serves as an introduction to the algorithms described in later chapters of this thesis. In Chapter 3, 4, and 5, a number of algorithms, applicable in a fault-tolerant distributed system based on system-level hardware masking redundancy, are presented. We will now give a brief overview of the contents of Chapters 3, 4, and 5.

Chapter 3 investigates so-called Byzantine Agreement Protocols (BAPs). BAPs solve the problem of reaching agreement among a set of correctly functioning processors despite the possible presence of a number of Byzantine faulty processors. This so-called agreement problem may occur in any fault-tolerant system with replicated system services due to broadcast errors exhibited by a faulty client machine communicating with one of the system services. Such a broadcast error may lead to a system breakdown, even if the system does not contain more faulty processors than it is designed to tolerate.

A BAP consists of a multicast process and a decision-making process. In the multicast process, in a number of communication phases, a message value is transmitted from the source (i.e., the client) to all other processors in the system. After that, in the decision-making process, each processor decides individually on basis of the message values of the messages it has accepted during the multicast process.

BAPs may be based on either regular messages, or on so-called authenticated messages. The BAPs based on regular messages (referred to as non-authenticated BAPs) require a huge communication overhead which often inhibits any practical implementation of the protocol. The BAPs based on authenticated messages (also referred to as authenticated BAPs) require much less communication overhead than the non-authenticated BAPs. In authenticated BAPs, every correct processor in the system is assumed to possess an unforgeable signature, with which it signs every message it relays. Signing each message limits the possible fault behaviour of faulty processors, hence, simplifies the BAP, and, as a result, it reduces the required communication overhead. Although the correctness of authenticated BAPs is dependent on the unforgeability of the signatures of the correct processors, in practice, this is not a great problem, since the probability of a correct processor's signature being forged can be made arbitrarily small.

Although authenticated BAPs require much less communication overhead than non-authenticated BAPs, the required number of messages is still often too large to be practical. Many attempts have been made to reduce the communication overhead in a BAP. In Section 3.2. of this thesis, the required communication overhead in an authenticated BAP is reduced by reducing the size of the messages. This is achieved by encoding the messages into symbols of an erasure-correcting code instead of multicasting them in the multicast process.

A major disadvantage of most BAPs that appeared in literature thus far, is that they are only applicable in a lock-step synchronous system, in which all correct processors know the start of the BAP and start the BAP simultaneously. As a consequence, these BAPs are not applicable in a distributed system.

To overcome this problem, Cristian et al. have proposed a BAP in which the correct processors need not know the start of the BAP or start the BAP simultaneously, and which is applicable in any synchronous system, provided that the processor clocks of all correct processors are approximately synchronized at the start and during execution of the BAP. Achieving and maintaining processor clocks approximately synchronized requires a majority of the processors to be correct, and a fault-tolerant clock synchronization algorithm to be periodically executed.

However, protocol synchronization (i.e., approximately synchronous execution of the protocol) between all correct processors is sufficient for a deterministic authenticated BAP to guarantee Byzantine agreement in a synchronous system. Clock synchronization (as is required by all existing deterministic authenticated BAPs in literature thus far) is only a means to establish protocol synchronization. To show this, in Section 3.3. of this thesis, authenticated self-synchronizing BAPs are introduced. These BAPs guarantee Byzantine agreement in any synchronous system containing an arbitrary number of faulty processors, while allowing arbitrary clock skew between the clocks of the processors in the system. The required amount of protocol synchronization is established during execution of the BAP itself by means of diffusion induction.

A disadvantage of these authenticated self-synchronizing BAPs is that a large communication overhead is required. In the optimized authenticated self-synchronizing BAPs, the required communication overhead is reduced by reducing the number of receivers in each communication phase. This reduction in the communication overhead is obtained at the cost of increased protocol execution time.

Besides the above-described authenticated self-synchronizing BAPs, a number of other fault-tolerant protocols are introduced in this thesis.

Chapter 4 introduces distributed cryptographic function application protocols (DCFAPs). These protocols are designed to work in any synchronous system consisting of a set, *N*, of *N* processors, up to *T* of which may behave maliciously. DCFAPs establish application of a secret cryptographic function *F* on a certain commonly known piece of data, *B*, and guarantee agreement among the correct processors in the system about the result of application of *F* on *B*, provided that the number of faulty processors in the system is less than or equal to *T*. Each processor in the system is given the capability to perform only part of the function application. The capabilities are chosen such that it is computationally infeasible for a set of *T* or fewer colluding processors to generate or apply *F* without the help of one or more correct processors.

Two different types of DCFAPs are introduced: interactive DCFAPs based on a fault-tolerant version of a multisignature scheme, and non-interactive DCFAPs based on function sharing. Both DCFAPs require at least a majority of the processors to be correct (i.e., $N \geq 2T+1$), in interactive DCFAPs, the required number of processors in the system may even be as high as T^2+T+1, depending on which component cryptographic key distribution has been selected.

It is shown that DCFAPs can conveniently be used as part of a recovery process responsible for recovery of lost or corrupted data fragments in a data storage system in which data is stored in a fault-tolerant and secure way on a set, *N*, of *N* processors, up to *T* of which may behave maliciously.

Finally, Chapter 5 describes a new Byzantine fault-tolerant distributed diagnosis algorithm. Just like the previously described protocols, this algorithm has also been designed to work in any synchronous system consisting of a set, *N*, of *N* processors, up to *T* of which may behave maliciously. The algorithm is executed independently by each processor. Execution of the diagnosis algorithm yields a set, *S*, of processors, which is a superset of the set of all detectably faulty processors in the system. It is of key importance that all faulty processors are removed as soon as possible from the system. Removing all processors of set *S* from the system guarantees that the remaining system contains correctly functioning processors only.

Set *S* is determined on basis of a set of test results, which is assumed to be agreed upon by and available in all correct processors prior to execution of the diagnosis algorithm. These test results are obtained by having every processor test every other processor and communicate the result of these tests to all other processors by means of a BAP. The tests are allowed to have an imperfect fault coverage, which is, in practice, almost always the case. Because tests may have an imperfect fault coverage, it is sometimes inevitable to include in set *S*, besides all detectably faulty processors, a number of correct processors too.

Many diagnosis algorithms have appeared in literature. Roughly, they can be divided into two groups: deterministic and probabilistic diagnosis algorithms, respectively. Deterministic diagnosis algorithms are guaranteed to find the set of all faulty processors, provided that tests have perfect fault coverage. This assumption is not very realistic. Therefore, a group of probabilistic diagnosis algorithms has appeared, in which tests are allowed to have only imperfect fault coverage, however, these algorithms only find the set of all faulty processors with a certain probability.

Our new diagnosis algorithm combines the best of both worlds by yielding a set which includes all detectably faulty processors, while allowing tests to have an imperfect fault coverage.

In conclusion, this thesis describes several new classes of Byzantine fault-tolerant algorithms for distributed systems based on system-level hardware masking redundancy. Application of these algorithms is of key importance in systems in which a high reliability is required and in which a limited number of processors is allowed to behave maliciously.

Samenvatting

Het steeds toenemend gebruik van computers in de hedendaagse maatschappij zorgt voor een toenemende afhankelijkheid van het correct functioneren van computersystemen. Hoewel de technische kwaliteit van de toegepaste hardwarecomponenten voortdurend wordt verbeterd kan het uitvallen van een computersysteem nooit helemaal worden voorkomen. De kans op het uitvallen van het systeem kan echter wel sterk worden verkleind door het toepassen van technieken van foutpreventie, foutdetectie, en fout-tolerantie in het systeem. Toepassing van elk van deze technieken is onontbeerlijk in kritische toepassingen die een hoge betrouwbaarheid vereisen, d.w.z., in systemen waarin de kans op het uitvallen van het systeem zo klein mogelijk moet zijn.

Systemen die zijn gebaseerd op foutpreventie en / of foutdetectie voldoen niet langer aan de systeemspecificatie zodra één of meerdere componenten van het systeem uitvallen. Een fout-tolerant systeem daarentegen kan aan zijn specificatie blijven voldoen, zelfs als één of meerdere componenten van het systeem zijn uitgevallen. Voor dit doel bevat elk fout-tolerant systeem één of andere vorm van redundantie, d.w.z., het systeem bevat componenten (redundantie in hardware) en / of voert berekeningen uit (redundantie in software) die overbodig zijn zolang er geen fouten optreden. Toepassing van redundantie in software kan een systeem bestand maken tegen transiënte en intermitterende fouten, echter, redundantie in hardware is vereist in elk systeem dat bestand moet zijn tegen permanente fouten. Redundantie in hardware kan op verschillende niveaus worden toegepast, bijv. op poortniveau of op systeemniveau.

Dit proefschrift behandelt het ontwerp van nieuwe algoritmen voor fout-tolerante systemen gebaseerd op het maskeren van fouten op systeemniveau d.m.v. redundantie in hardware. Er wordt beargumenteerd dat ieder systeem waarin een betrouwbaarheidsverbetering van tenminste een factor 100 is vereist gebaseerd dient te zijn op het maskeren van fouten op systeemniveau d.m.v. redundantie in hardware. De techniek van het maskeren van fouten op systeemniveau d.m.v. redundantie in hardware is toepasbaar in een redundant systeem, bestaande uit een aantal processoren, waarin de systeemservices gerepliceerd zijn op de verschillende processoren, en waarin een beperkt aantal processoren zich fout mag gedragen. De techniek is het meest effectief in een gedistribueerd systeem, omdat de autonome aard en de geografische spreiding van de processoren in een dergelijk systeem sterk bijdragen aan onafhankelijkheid tussen het uitvallen van verschillende processoren in het systeem, hetgeen de betrouwbaarheid van het systeem ten goede komt.

Een gedistribueerd systeem bestaat uit een aantal processoren dat met elkaar verbonden is d.m.v. een netwerk. Elke processor heeft zijn eigen geheugen en zijn eigen processorklok. Processoren in een gedistribueerd systeem kunnen slechts informatie uitwisselen d.m.v. het sturen van berichten naar elkaar.

Een gedistribueerd systeem kan worden gemodelleerd als een synchroon systeem. In

dit proefschrift zal een synchroon systeem worden gedefinieerd als een systeem met synchrone communicatie (d.w.z., we veronderstellen het bestaan van een real-time niet-negatieve bovengrens τ_{max} op de tijd die nodig is om een bericht van een correcte processor naar een andere processor in het systeem te sturen) en synchrone processoren (d.w.z., de graad waarmee de processorklok van een willekeurige correcte processor afwijkt van real-time is begrensd door een factor $(1+\rho)$, voor bepaalde $\rho \geq 0$). Tevens wordt het bestaan verondersteld van een real-time niet-negatieve ondergrens τ_{min} (met $\tau_{min} \leq \tau_{max}$) op de tijd die nodig is om een bericht van een correcte processor naar een andere processor te sturen, en er wordt aangenomen dat de grenzen τ_{min}, τ_{max}, en $(1+\rho)$ bekend zijn bij elke correcte processor in het systeem. In de praktijk bestaat er in een gedistribueerd systeem altijd enige onzekerheid over de tijdsduur die nodig is voor het oversturen van een bericht (d.w.z., $\tau_{min} \neq \tau_{max}$), en bestaan er geen twee processorklokken die precies even snel lopen (d.w.z., $\rho > 0$).

De processoren in een gedistribueerd systeem zijn autonoom in die zin, dat er geen gemeenschappelijk geheugen en dus geen a priori globale kennis aanwezig is tussen de processoren in het systeem. De processoren kunnen slechts informatie uitwisselen d.m.v. het zenden van berichten naar elkaar. Met name is het in een gedistribueerd systeem onmogelijk om alle correcte processoren precies tegelijk een gedistribueerd algoritme te laten opstarten, omdat het onmogelijk is om alle correcte processoren overeenstemming te laten bereiken over een gezamenlijk tijdstip, immers, iedere processor heeft zijn eigen processorklok, er bestaan geen twee processorklokken die precies even snel lopen, en elk willekeurig fout-tolerant kloksynchronisatie-algoritme kan de klokken van de correcte processoren in het systeem slechts bij benadering synchroniseren.

Veel fout-tolerante algoritmen uit de literatuur vereisen een zogenaamd lock-step synchroon systeem. Een lock-step synchroon systeem is een synchroon systeem waarin $\tau_{min} = \tau_{max}$ en $\rho=0$. Verder eisen deze algoritmen dat alle correcte processoren weten wanneer het algoritme begint en dat alle correcte processoren het algoritme tegelijkertijd beginnen. Omdat in een gedistribueerd systeem aan geen enkele van de bovengenoemde eisen kan worden voldaan, kunnen deze algoritmen niet worden toegepast in een gedistribueerd systeem.

Vrijwel alle algoritmen[1] beschreven in dit proefschrift zijn toepasbaar in een willekeurig synchroon systeem (d.w.z., een synchroon systeem met willekeurige, bekende niet-negatieve ρ, τ_{min}, en τ_{max}, met $\tau_{min} \leq \tau_{max}$) bestaande uit een verzameling, N, van N processoren, waarin de systeemservices gerepliceerd zijn op de verschillende processoren in het systeem. De algoritmen zijn zelfsynchroniserend, d.w.z., het is niet vereist dat de correcte processoren weten wanneer het algoritme begint en het algoritme tegelijkertijd beginnen. De algoritmen voldoen aan hun specificatie zolang het aantal foute processoren in het systeem kleiner of gelijk is aan een of andere van tevoren gedefinieerde systeemparameter, T.

In het algemeen kunnen processoren op een willekeurige manier falen. Helaas is het in

1. Alleen de algoritmen beschreven in paragraaf 3.2. vereisen een lock-step synchroon systeem, waarin alle correcte processoren weten wanneer het algoritme begint en het algoritme tegelijkertijd opstarten.

de praktijk onmogelijk om het systeem bestand te maken tegen willekeurige fouten van zijn processoren. Om dit probleem te omzeilen wordt verondersteld dat de processoren falen overeenkomstig een bepaald van tevoren gedefinieerd foutmodel. Hoe zwakker het foutmodel (d.w.z., hoe algemener het foutgedrag dat is toegestaan in het foutmodel), des te hoger is de betrouwbaarheid van het systeem. Iedere willekeurige foute processor in het systeem wordt verondersteld te falen overeenkomstig een bepaald begrensd Byzantijns foutmodel. Het in dit proefschrift gekozen foutmodel omvat vrijwel ieder foutgedrag dat relevant is voor de beschouwde algoritmen.

Dit proefschrift is als volgt opgebouwd.

Hoofdstuk 1 begint met een inleiding in dependable computing.

Daarna wordt in hoofdstuk 2 het zogenaamde Dependable Distributed Data Storage (DDDS) systeem gepresenteerd. Dit is een virtueel systeem voor betrouwbare gegevensopslag, waarin de in dit proefschrift beschreven algoritmen handig kunnen worden toegepast. Het DDDS systeem dient als inleiding op de in de latere hoofdstukken van dit proefschrift beschreven algoritmen. In hoofdstuk 3, 4, and 5 van dit proefschrift wordt een aantal algoritmen gepresenteerd, dat toepasbaar is in een fout-tolerant gedistribueerd systeem gebaseerd op het maskeren van fouten op systeem-niveau d.m.v. redundantie in hardware. We zullen nu kort ingaan op de inhoud van hoofdstuk 3, 4, en 5.

Hoofdstuk 3 behandelt zogenaamde Byzantijnse Generaals Algoritmen (BGA’s). BGA’s vormen een oplossing voor het probleem van het bereiken van overeenstemming binnen een verzameling correct functionerende processoren, ondanks de mogelijke aanwezigheid van een aantal zich Byzantijns fout gedragende processoren. Dit zogenaamde agreement-probleem kan optreden in elk willekeurig fout-tolerant systeem met gerepliceerde systeemservices als gevolg van broadcastfouten veroorzaakt door een met één van de systeemservices communicerende clientmachine. Een dergelijke broadcast error kan leiden tot een breakdown van het hele systeem, zelfs als het systeem niet méér foute processoren bevat dan dat volgens het systeemontwerp toegestaan zijn.

Een BGA bestaat uit een multicast-proces en een beslissingsproces. In het multicast-proces wordt in een aantal communicatiefasen een bericht uitgezonden door de source naar alle andere processoren in het systeem. In het erop volgende beslissingsproces beslist iedere processor individueel op basis van de zogenaamde message value van de berichten (messages) die hij heeft geaccepteerd tijdens het multicast-proces.

BGA’s kunnen gebaseerd zijn op reguliere berichten of op zogenaamde geauthenticeerde berichten. De op reguliere berichten gebaseerde BGA’s (ook wel ongeauthenticeerde BGA’s genaamd) vereisen een geweldige communicatie-overhead, die meestal elke praktische implementatie van het algoritme in de weg staat. De BGA’s gebaseerd op geauthenticeerde messages (ook wel geauthenticeerde BGA’s genoemd) vereisen veel minder communicatie-overhead dan de ongeauthenticeerde BGA’s. In geauthenticeerde BGA’s wordt verondersteld dat iedere correcte processor in het systeem beschikt over een onvervalsbare digitale handtekening, waarmee hij elk bericht dat hij doorstuurt, tekent. Het tekenen van berichten perkt het mogelijke foutgedrag van foute processoren in, als zodanig vereenvoudigt dit de benodigde BGA en reduceert dit de benodigde communicatie-overhead. Hoewel de correctheid van geauthenticeerde BGA's afhangt van de onvervalsbaarheid van de handtekeningen van de correcte processoren, is dit in de praktijk geen groot probleem, omdat de kans op het vervalsen van een handtekening willekeurig klein gemaakt kan worden.

Ofschoon geauthenticeerde BGA's veel minder communicatie-overhead vereisen dan ongeauthenticeerde BGA's is het benodigd aantal berichten vaak nog steeds te groot voor een praktische toepassing. Er zijn vele pogingen gedaan om de communicatie-overhead in een BGA te beperken. In sectie 3.2. van dit proefschrift wordt de communicatie-overhead beperkt door de grootte van de berichten te reduceren. Dit wordt bewerkstelligd door de berichten te coderen in symbolen van een fouten-corrigerende code in plaats van ze te multicasten in het multicast-proces.

Een groot nadeel van de meeste tot nu toe in de literatuur verschenen BGA's is dat ze alleen toepasbaar zijn in een lock-step synchroon systeem, waarin alle correcte processoren de start van het BGA weten en het BGA tegelijkertijd opstarten. De consequentie hiervan is dat deze BGA's niet toepasbaar zijn in een gedistribueerd systeem.

Om dit probleem op te lossen hebben Cristian e.a. een BGA geïntroduceerd waarin de correcte processoren niet hoeven te weten wanneer het BGA begint en evenmin het BGA tegelijkertijd hoeven op te starten. Dit BGA is toepasbaar in een willekeurig synchroon systeem, mits de processorklokken van de correcte processoren vrijwel gelijklopen (d.w.z., binnen van tevoren bekende grenzen gesynchroniseerd zijn) aan het begin en tijdens uitvoering van het BGA. Vrijwel gelijklopende processorklokken kunnen worden verkregen door het periodiek uitvoeren van een fout-tolerant kloksynchronisatie-algoritme, mits een meerderheid van de processoren correct functioneert.

Echter, protocolsynchronisatie (d.w.z., vrijwel gelijktijdige executie van het BGA) tussen alle correcte processoren is reeds voldoende voor een deterministisch geauthenticeerd BGA om in een synchroon systeem Byzantine Agreement te kunnen bereiken. Kloksynchronisatie (zoals vereist is in alle deterministische geauthenticeerde BGA's in de literatuur tot nu toe) is slechts een middel om protocolsynchronisatie tot stand te brengen. Dit wordt getoond in sectie 3.3. van dit proefschrift d.m.v. de introductie van geauthenticeerde zelfsynchroniserende BGA's. Deze BGA's garanderen Byzantine Agreement in een willekeurig synchroon systeem waarin een willekeurig aantal processoren mag falen, terwijl een willekeurige clock skew tussen de klokken van de processoren in het systeem is toegestaan. De vereiste hoeveelheid protocolsynchronisatie wordt verkregen tijdens het uitvoeren van de BGA d.m.v. diffusion induction.

Een nadeel van deze geauthenticeerde zelfsynchroniserende BGA's is dat een grote communicatie-overhead vereist is. In de geoptimaliseerde geauthenticeerde zelfsynchroniserende BGA's wordt de vereiste communicatie-overhead verkleind door vermindering van het aantal ontvangers in elke communicatiefase. Vermindering van de communicatie-overhead gaat gepaard met een langere executietijd van het algoritme.

Naast de zojuist beschreven geauthenticeerde zelfsynchroniserende BGA's wordt in hoofdstuk 4 en 5 van dit proefschrift nog een aantal andere fout-tolerante protocollen geïntroduceerd.

In hoofdstuk 4 worden de zogenaamde distributed cryptographic function application protocols (DCFAPs) geïntroduceerd. Deze protocollen werken in een willekeurig synchroon bestaande uit een verzameling, $\mathcal{N}$, van N processoren, waarvan er ten hoogste T zich fout mogen gedragen. DCFAPs zorgen ervoor dat een geheime cryptografische functie F wordt toegepast op een bepaald gemeenschappelijk stuk data, B, en garanderen overeenstemming tussen de correcte processoren in het systeem over het resultaat van toepassing van F op B, mits het aantal foute processoren in het systeem kleiner of gelijk is aan T. Elke processor in het systeem is in staat om een deel van de functie-applicatie te doen. De capaciteiten zijn zodanig gekozen dat het voor een verzameling van T of minder samenspannende processoren onmogelijk is om F te genereren of toe te passen zonder de hulp van één of meer correcte processoren.

Er worden twee verschillende typen DCFAPs geïntroduceerd: interactieve DCFAPs gebaseerd op een fout-tolerante versie van een multisignature scheme, en non-interactieve DCFAPs gebaseerd op function sharing. Beide typen DCFAPs vereisen dat tenminste een meerderheid van de processoren correct is (d.w.z., $N \geq 2T+1$), in interactieve DCFAPs kunnen zelfs $T^2 + T + 1$ processoren vereist zijn in het systeem, afhankelijk van welke component cryptographic key verdeling geselecteerd is.

Er wordt aangetoond dat DCFAPs handig gebruikt kunnen worden als onderdeel van een recovery-proces dat verantwoordelijk is voor recovery van verloren gegane of gecorrumpeerde datafragmenten in een data-opslagsysteem waarin data op een fout-tolerante en secure manier is opgeslagen op een verzameling, $\mathcal{N}$, van N processoren, waarvan er zich ten hoogste T fout mogen gedragen.

Hoofdstuk 5 tenslotte beschrijft een nieuw Byzantijns fout-tolerant gedistribueerd diagnose-algoritme. Net als de eerder beschreven protocollen is ook dit algoritme geschikt voor een willekeurig synchroon systeem bestaande uit een verzameling, $\mathcal{N}$, van N processoren, waarvan er zich ten hoogste T fout mogen gedragen. Het algoritme wordt onafhankelijk uitgevoerd door elke processor. Het uitvoeren van het diagnose-algoritme levert een verzameling, S, van processoren op, die een superset is van de verzameling van alle detecteerbaar foute processoren. Het is van wezenlijk belang om alle foute processoren zo snel mogelijk uit het systeem te verwijderen. Het verwijderen van alle processoren in verzameling S uit het systeem garandeert dat het overblijvende systeem alleen maar correct functionerende processoren bevat.

Verzameling S wordt bepaald op basis van een verzameling van testresultaten, waarover de correcte processoren overeenstemming bereikt hebben en die beschikbaar is in alle correcte processoren vóór uitvoering van het diagnose-algoritme. Deze testresultaten worden verkregen door elke processor elke andere processor te laten testen en de resultaten van deze tests te communiceren naar alle andere processoren d.m.v. een BGA. De tests mogen een imperfecte fout-coverage hebben; dit is in de praktijk bijna altijd het geval. Doordat het is toegestaan dat tests een imperfecte fout-coverage hebben is het soms onvermijdelijk om naast alle detecteerbaar foute processoren ook een aantal correcte processoren in verzameling S op te nemen.

Er zijn in de literatuur veel diagnose-algoritmen verschenen. Grofweg kunnen deze algoritmen in twee groepen worden verdeeld: deterministische en probabilistische diagnose-algoritmen, resp. Deterministische diagnose-algoritmen vinden gegarandeerd de verzameling van alle foute processoren, mits de tests een perfecte fout-coverage hebben. Deze aanname is niet erg realistisch. Daartoe is een groep probabilistische diagnose-algoritmen ontwikkeld, waarin tests een imperfecte fout-coverage mogen hebben. Deze algoritmen vinden de verzameling van alle foute processoren echter slechts met een bepaalde waarschijnlijkheid.

Ons nieuwe diagnose-algoritme combineert de voordelen van beide alternatieven door een verzameling op te leveren waarin alle detecteerbaar foute processoren bevat zijn, terwijl het toegestaan is dat tests slechts een imperfecte fout-coverage hebben.

Resumerend beschrijft dit proefschrift verscheidene nieuwe klassen van Byzantijns fout-tolerante algoritmen, geschikt voor gedistribueerde systemen gebaseerd op het maskeren van fouten op systeemniveau d.m.v. redundantie in hardware. Toepassing van deze algoritmen is van cruciaal belang in systemen waarin een hoge betrouwbaarheid vereist is, en waarin een beperkt aantal willekeurige processoren zich fout mag gedragen.

List of Publications

[PBHS96] Postma, A., Boer, W. de, Helme, A., and Smit, G., **Distributed Encryption and Decryption Algorithms**, Memoranda Informatica 96-20, University of Twente, Enschede, December 1996.

[PKrM97] Postma, A., Krol, Th., and Molenkamp, E., Optimized Authenticated Self-synchronizing Byzantine Agreement Protocols, to appear in: **Proceedings of the 1997 Pacific Rim International Symposium on Fault-Tolerant Systems (PRFTS'97)**,Taipei, Taiwan, December 15-16, 1997. (An extended version of this paper appeared as: CTIT Technical Report 97-20, University of Twente, Enschede, September 1997.)

[PoHK96] Postma, A., Hartman, G., and Krol, Th., Removal of All Faulty Nodes from a Fault-Tolerant Service by Means of Distributed Diagnosis with Imperfect Fault Coverage, in: **Dependable Computing - EDCC-2, Second European Dependable Computing Conference, Taormina, Italy, October 1996, Proceedings**, Hlawiczka, A., Silva, J.G., and Simoncini, L. (Eds.), Springer-Verlag, Berlin, LNCS 1150, 1996, pp.385-402.

[PoKM97] Postma, A., Krol, Th., and Molenkamp, E., Distributed Cryptographic Function Application Protocols, in: Han, Y., Okamoto, T., and Qing, S. (Eds.), **Information and Communications Security, First International Conference, ICICS'97, Beijing, China, November 1997, Proceedings**, LNCS 1334, Springer-Verlag, Berlin, 1997, ISBN 3-540-63696-X, pp.435-439.

[PoKr95] Postma, A., and Krol, Th., Interactive Consistency Algorithms Based on Authentication and Error-Correcting Codes, in: **Proceedings of the 5th International Working Conference on Dependable Computing for Critical Applications DCCA-5**, Urbana-Champaign, Illinois, 1995, preprints, pp.91-101.

[PoKr96] Postma, A., and Krol, Th., Interactive Consistency in Quasi-Asynchronous Systems, in: **Proceedings of the 2nd IEEE International Conference on Engineering of Complex Computer Systems ICECCS'96**, Montréal, Canada, October 21-25, 1996, pp.2-9. (A preliminary version of this paper appeared as: Memoranda Informatica 96-05, University of Twente, Enschede, March 1996.)

[Post96] Postma, A., **Implementation of a simulation program of a self-synchronizing Byzantine Agreement Protocol**, Memoranda Informatica 96-19, University of Twente, Enschede, November 1996.

Acknowledgements

This research was carried out as a part of the FADE-project (FAult-tolerant DEsign) in the group Systems Programming and Architecture at the Department of Computer Science at the University of Twente in Enschede.

In this place, I would like to thank all people that have contributed in one way or another to this thesis. In particular I would like to thank

❏ my promotor Thijs Krol for his numerous suggestions and for the opportunity to work in the group Systems Programming and Architecture.

❏ Bert and Giny Molenkamp for their support and continuous interest, and for making me feel at home in Hengelo. I would also like to thank Bert for his many valuable suggestions to improve my thesis, and for the many pleasant walks around the campus.

❏ my roommate Gerhard Mekenkamp for being a good pal, for his continuous interests in my work and my personal life, for the numerous discussions about a wide variety of topics, and also for the many pleasant walks around the campus.

❏ Corrie Huijs for her great support during the writing of this thesis.

❏ Jan Wessels, Gerie Hartman, and Willem de Boer for sharing their ideas with me. Especially, I would like to thank Gerie Hartman for all her efforts and enthousiasm, which resulted in a joint paper on a conference in Taormina, Sicily. Chapter 5 of this thesis has been based on this paper.

❏ Gert and Inge Postma for being my ‘family’ in Hengelo, for taking good care of my apartment every time I visited a conference.

❏ Joost-Anne and Tiny Veerman, Harry van der Pol, Cor-Jan van de Veen, Ina Tolman, Peter and Daniëlle van Leeuwenstijn, Hendrik and Simone de Jong, Louise Ellenkamp, Martijn Meupelenberg, and Hans and Petra Hiemstra, for being close friends, for their continuous interest in my personal life and for showing me that life is more than work alone.

❏ my parents and my brother Peter for all their support.

Last but not least, I would like to thank all friends and colleagues which have not been mentioned above, with which I shared the last four years in Twente.

Biography

André Postma was born in Aalst-Waalre, the Netherlands, on 19 August 1968. He obtained his VWO-diploma from the Grotius College in Heerlen in 1986.

Subsequently, he studied Computer Science at the Hogeschool Heerlen, Sector Techniek, from which he graduated in 1990. He continued his study of Computer Science at the Eindhoven University of Technology, from which he graduated in computer graphics in 1993 on a master’s thesis entitled: ‘Geometric modeling by object substitution’.

Since 1994, after having fulfilled his military service as a truck driver in Seedorf, Germany, he has worked as a Ph.D. student under the supervision of Prof.Dr.Ir. Thijs Krol, in the group Systems Programming and Architecture, of the Computer Science Department of the University of Twente in Enschede. He has worked in the FADE-project (FAult-tolerant DEsign) on the design of Byzantine fault-tolerant algorithms for dependable distributed systems. This research has resulted in this Ph.D.thesis.

At 1 March 1998 he will start to work at Philips Research Laboratories in Eindhoven.

'Doing research is about science.
Finishing a thesis is about emotions.'

J.M. Nauta - Vehicle Trajectory Planning with Interval Arithmetic

Stellingen

behorende bij het proefschrift

'Classes of Byzantine Fault-Tolerant Algorithms for Dependable Distributed Systems'

door

André Postma

20 februari 1998

1. Het is merkwaardig dat er geen consensus bestaat over de definitie ervan in de literatuur.

   (Hoofdstuk 3 van dit proefschrift)

2. In Byzantijnse Generaals algoritmen voor synchrone systemen is protocolsynchronisatie vereist; gesynchroniseerde processorklokken zijn slechts een hulpmiddel om de vereiste protocolsynchronisatie te bereiken.

   (Hoofdstuk 3 van dit proefschrift)

3. Protocolsynchronisatie in een deterministisch geauthenticeerd Byzantijns Generaals algoritme kan tot stand gebracht worden tijdens executie van het algoritme zelf, zonder gebruik te maken van gesynchroniseerde processorklokken. Het op deze manier tot stand brengen van protocolsynchronisatie, zoals is gedaan in de in dit proefschrift geïntroduceerde geauthenticeerde zelfsynchroniserende Byzantijnse Generaals algoritmen, maakt deze algoritmen geschikt voor toepassing in een gedistribueerd systeem.

   (Hoofdstuk 3 van dit proefschrift)

4. Onder de aanname dat tests die gebruikt worden voor diagnose van processoren in een gedistribueerd systeem een imperfecte fout-coverage hebben, kan het op basis van de testresultaten onmogelijk zijn om te beslissen of een processor zich al dan niet correct gedraagt. Onder de betreffende aanname bestaat er derhalve geen diagnose-algoritme dat altijd de verzameling van *precies* alle zich niet correct gedragende processoren oplevert. Wordt echter naast de bovengenoemde aanname tevens verondersteld dat elke foute processor detecteerbaar fout gedrag vertoont, dan bestaat er een niet triviaal diagnose-algoritme dat een verzameling van processoren oplevert die altijd *tenminste* alle zich niet correct gedragende processoren bevat, en waarbij het totaal aantal processoren in die verzameling altijd kleiner of gelijk is aan $(T + 2)^2 / 4$, waarbij T het maximaal aantal foute processoren in het systeem is.

   (Hoofdstuk 5 van dit proefschrift)

5. Het woord “proefschrift” kan beter vervangen worden door “schrijfproef”.

6. Dubbel parkeren als oplossing voor het parkeerprobleem is in overeenstemming met het feit dat fout-tolerante oplossingen altijd duur zijn.

7. Het gezegde ‘Wie schrijft die blijft’ is met de huidige bezuinigingen op universiteiten en het aantrekken van de arbeidmarkt voor informatici, niet van toepassing op informatica-AiO’s, aangezien het merendeel tijdens of na het schrijven van zijn/haar proefschrift de universiteit verlaat.

8. In militaire dienst leer je onder andere afzien en omgaan met teleurstellingen. Het is een goede voorbereiding op het AiO-schap.

9. De invoering van het groene boekje voor de spelling van de Nederlandse taal zorgt ervoor dat nu ook degenen die nog niet twijfelden aan de juiste spelling van het Nederlands, aan het twijfelen worden gebracht.

10. Op een fout-tolerant ontwerp kan beter geen copyright zitten.