DISSERTATION

Authenticated Consensus

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

Univ.Prof. Dr. Ulrich Schmid

Inst.-Nr. E183/1
Institut für Automation

eingereicht an der Technischen Universität Wien
Technisch-Naturwissenschaftliche Fakultät

von

Bettina Weiss

Matr.-Nr. 8926046

Feldgasse 21
A-2542 Kottingbrunn

Wien, im Oktober 2002 ______________________

Zusammenfassung

Diese Arbeit befaßt sich mit authentifiziertem Consensus in drahtlosen Netzwerken und entstand im Kontext des Projekts W2F. W2F ist der Entwicklung eines Feldbusses gewidmet, der drahtlose und drahtgebundene Kommunikation nahtlos integrieren und den Aufbau eines verteilten, fehlertoleranten und sicheren Netzwerks trotz flexibler dezentraler Netzwerktopologie und mobilen Teilnehmern erlauben soll. Consensus wird zur Entscheidungsfindung innerhalb einer Gruppe von Knoten (Computer) benötigt und ist daher eine wesentliche Grundlage jedes redundanten verteilten Systems und somit auch Bestandteil von W2F. Mittels Authentifizierung kann die Identität des Senders einer Nachricht festgestellt werden. Durch sie werden Angreifer daran gehindert, Schaden im Namen anderer Knoten anzurichten, was die Lösung des Consensus-Problems generell erleichtert und in W2F mit seiner flexiblen und dynamischen Topologie besonders wichtig ist.

Der erste Teil der Arbeit ist der Authentifizierung mittels digitaler Signaturen und ihren Problemen in W2F gewidmet. Dabei betrachten wir vor allem sogenannte denial-of-service Angriffe, bei denen der Angreifer versucht, die Resourcen des angegriffenen Knotens zu erschöpfen. Um potentielle Angriffe möglichst zu verhindern und den zeitlichen Aufwand des angegriffenen Knotens zu minimieren, verwenden wir Message Authentication Codes und substantielle Hardwareunterstützung in Form von Nachrichtenfiltern. Weiters integrieren unsere Algorithmen Techniken, mit denen Angreifer dazu gezwungen werden, ihrerseits beträchtlichen Aufwand zu leisten, um den angegriffenen Knoten zu einer aufwendigen Tätigkeit zu veranlassen.

Im zweiten Teil geht es um Consensus in W2F. Neben der Präsentation des in W2F verwendeten hybriden Fehlermodells werden untere Schranken für die Mindestzahl von Knoten hergeleitet, die man benötigt, um Consensus in unserem Fehlermodell zu erreichen. Danach folgt die Analyse von mehreren Byzantine-Agreement Algorithmen, mit denen man Consensus erzielen kann. Neben dem wohlbekannten unauthentifizierten Algorithmus OMH (Lincoln & Rushby, 1993) analysieren wir zwei authentifizierte Varianten OMHA und ZA (Gong, Lincoln & Rushby, 1995). Unsere Ergebnisse zeigen, daß Consensus in Systemen mit dominanten Link-Fehlern lösbar ist, ungeachtet des wohlbekannten Unmöglichkeitsresultats von Jim Gray aus 1978.

Abstract

This thesis is concerned with authenticated consensus and has been developed in the context of project W_2F, which aims towards integrating wireless and wireline communication in a distributed fault-tolerant and secure fieldbus with ad-hoc (i.e., flexible, decentralized and unknown) network topology and mobile nodes. Consensus is a well-known problem in distributed systems and is used to find a common decision among a group of nodes. This is a fundamental building block of redundant systems and thus a vital part of W_2F. Authentication is used to determine the identity of the sender of a message, even if the message has been passed along by other nodes. Attackers are thus deprived of the capability to mask their activities under cover of different identities, which in general simplifies achieving consensus and is an important asset in an ad-hoc network like W_2F.

The first part of this thesis is dedicated to authentication with digital signatures and its special needs in W_2F. In order to avoid or at least diminish denial-of-service attacks and to minimize the cost to the attacked node, we use message authentication codes and employ substantial hardware support in the form of message filters. Our algorithms also integrate the client puzzle technique to make the attacker spend its time and resources before the attacked node does.

The second part addresses consensus in W_2F. We will present the failure model used in the W_2F network and give some lower bounds on the number of nodes required to mask link failures. We will then analyze several Byzantine agreement algorithms which can be used to achieve consensus. Apart from the well-known OMH algorithm (Lincoln & Rushby, 1993) we will also examine two authenticated variants OMHA and ZA (Gong, Lincoln & Rushby, 1995). As it turns out, it is indeed possible to achieve consensus in systems with dominant link failures, despite the well-known impossibility result of Jim Gray from 1978.

Acknowledgments

I would like to thank my advisor, Prof. Dr. Ulrich Schmid, for his support and guidance during the past years. His diligence and dedication to his work have been an inspiration to me, and it has always been a pleasure and honor to work with him. His insights into the core of problems, paired with his delight in toying with new ideas and tracking down difficult problems, have made all discussions with him both instructive and fun. The consensus part of this thesis is the result of our joint work.

I also wish to express my gratitude to Prof. Dr. Mehdi Jazayeri, the head of the Distributed Systems Group in the Information Systems Institute, for his valuable comments and suggestions when grading my thesis.

I appreciate the input of John Rushby, whose remarks on the problem of disclosing private keys during his visit at the TU Vienna led me to investigate the problem of broken signatures. Moreover, my colleagues in the W_2F project team, in particular Günther Gridling and Martin Biely, provided me with many insightful comments and ideas.

I would like to thank my parents, Alfred and Inge Weiss, for allowing me to pursue my studies in my own time and for supporting me throughout these years.

Last but certainly not least, my profound gratitude goes to my friend Günther, whose continued encouragement and support, help, and ideas, both professional and personal, have been invaluable to me.

This work is part of the W_2F-project (*http://www.auto.tuwien.ac.at/Projects/W2F/*), which received funding from the Austrian START-programme Y41-MAT.

Contents

Chapter 1

Introduction

1.1 Project W2F

The W2F project has been started at the Department of Automation in 1999, see *http://www.auto.tuwien.ac.at/Projects/W2F/*. It aims to design a distributed, fault-tolerant and secure fieldbus seamlessly integrating both wireline and wireless communication. W2F will use spread-spectrum CDMA as its communication mechanism and will either provide its fault-tolerance, security and real-time properties even under failure or attack conditions or at least degrade gracefully. Its three major areas of research are:

Higher-Level Protocols: W2F will be equipped with suitable protocols for clock synchronization, secure connection management, reliable at-most-once data transmission, group management, and atomic multicasting.

System Architecture: The envisioned protocols need certain hardware support for low-level security mechanisms and synchronized clocks. The node architecture must also accommodate secure co-processors.

CDMA Wireless Spread-Spectrum Communications: Our primary goal here is to avoid the real-time communications scheduling problem required for all shared channels by using CDMA spread-spectrum communications both in wireline and wireless networks.

We envision the W2F network to be distributed with no base stations, i.e., to have an ad-hoc nature with no fixed topology, and to admit topology changes during operation [54]. We assume that all communication in the system, both wireline and wireless, uses spread-spectrum CDMA on top of UWB [21]. Spread-spectrum techniques use either a pseudo-noise sequence or some specially designed code sequence to spread every data bit of a message over the whole frequency band. To this aim, the sequence, also called the *CDMA code*, is multiplied with each data bit in turn, generating the transmit message. The encoded message is either broadcast as is (base-band system) or modulated on a carrier frequency to use some other frequency band.

The appealing property of CDMA lies in the fact that the spread signal can only be recovered by nodes possessing the proper CDMA code, whereas all other nodes experience the transmission as noise. Still, the CDMA code cannot be used as a shared secret key in W2F: Although messages cannot be deciphered without the proper code, every transmission acts as noise and thereby deteriorates the signal quality of other communication traffic. The farther away the recipient of a transmission, the more noise the transmission produces for nodes in the vicinity of the sender. Without proper counteractive measures, only a handful of simultaneous short-range transmissions per node would be possible [68, 28]. To allow for more and farther transmissions, however, nodes have to employ multi-user interference reduction techniques, which, unfortunately, require the CDMA codes of interfering transmissions to work effectively. Therefore, although tempting, we must refrain from relying on the CDMA codes for authenticated and encrypted communication and have to employ conventional mechanisms instead.

Due to the limits on simultaneous transmissions imposed by the CDMA technology, the W2F network will be only sparsely connected. We cannot expect every node to have a direct link to every other node. Instead, connectivity will be very good in areas of high node density and will be poor in regions of low population. Yet, for fault-tolerance reasons and to avoid network partitions, we must request that every node is—directly or indirectly over relay nodes—connected via k node-disjoint paths to every other node in the network. In areas of high density we can even assume a fully connected subnetwork, which can either be point-to-point or a broadcast channel. Figure 1.1 depicts a possible W2F network with $k = 2$.

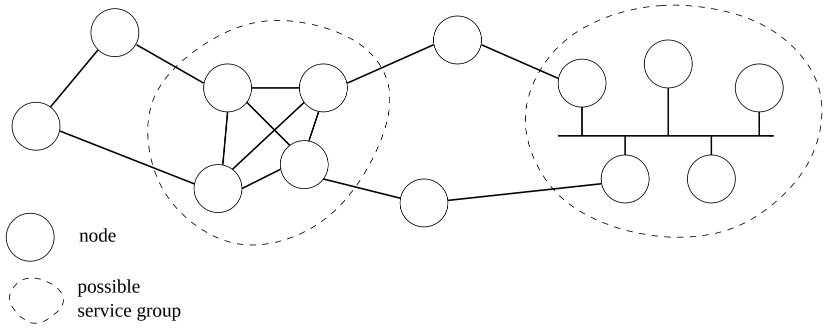

Figure 1.1. *A W2F network with areas of high and low node density.*

This thesis is concerned with authenticated consensus, which is used to reach agreement among a group of nodes. Since consensus and other distributed algorithms profit from or even require high connectivity, we expect that these algorithms will mainly be executed on nodes in the high density areas. These nodes will form service groups and provide services to nodes in the low density regions. Within the service groups, we can assume a fully connected network and a synchronous system, i.e., we can give bounds on transmission delays, clock drifts, and computation times. Outside of the group, such bounds may be significantly larger or not available at all. In order to model the behavior of the W2F network realistically, we use a hybrid failure model and pay particular attention to link failures, which may occur with a probability of up to and exceeding 10^{-2} in wireless settings.

In the authentication part of the thesis, we must again pay attention to the needs of our W2F network. Due to its dynamic nature, the network is particularly vulnerable to mobile intruders launching denial-of-service attacks, hence we must minimize the opportunities for successfully launching such attacks against our authentication algorithms.

The authentication algorithms have been published in [73], a more thorough treatment of the subject can be found in the corresponding technical report [74]. The analysis of the Byzantine agreement algorithms has been published in [59] and in [75]. The theoretical results concerning consensus under link failures can be found in [58].

1.2 Related Work

Given the importance of denial-of-service attacks, this problem has already been investigated from different angles in the literature. Many papers specifically address solutions to the TCP SYN flooding attack [9], a frequently exploited vulnerability in the TCP/IP protocol which allows attackers to establish half-open connections with a server, causing the server to create in its memory data structures for pending connection requests which are never completed. Since the server's memory is finite and must hold the structures until a given timeout, and since the attacker may use a spoofed IP address to conceal its true identity, the server's memory may fill with these data structures and legitimate requests will be denied. After attacks in 1999 and again in 2000 on several large Internet sites including Yahoo and Amazon [24], efforts have increased even more to prevent such attacks in the future. The suggested solutions mainly aim at preventing address spoofing (e.g. through authentication) and tracing attack packets back to the sender [66], at alleviating their impact through random packet drop strategies [48], or at detecting and filtering out attack packets [72].

Zhou and Haas [79] consider security in ad-hoc networks. Their goal is to secure the routing service from attacks by using signatures and by relying on connection redundancy. For them, denial-of-service means that wrong routing information is injected; they do not consider a slow-down of the service. Zhang and Lee [77] deal with intrusion detection techniques in wireless ad-hoc networks. In their proposed architecture, every node participates in the effort to detect abnormal activities. They assume the existence of a secure communication channel among nodes.

Matsura and Imai [38] provide strategies for combating DoS attacks. Their basic approach is to employ a "falling-together" strategy where the attacker has to pay a computational cost that is comparable to that of the target. The attacker should also make the first move, i.e., be the first one to employ heavy computation which can easily be verified. Only after the attacker has shown its commitment by performing the required computations should the target do any heavy computations on its own part. The same approach is proposed by Aura et al. in [4], where the authors use client puzzles [32] to make the client commit its resources first. The protocol does not require the server to store any client-related state until the client has committed itself. An appealing feature of the approach is the ability of the server to choose the difficulty of the puzzle by a parameter sent with the challenge. This allows a server to react to increasing overload conditions by dynamically increasing the difficulty of the puzzles.

Meadows [39] proposes a framework for the analysis of denial-of-service attacks which allows the protocol designer to compute the costs of such an attack to the defender and to compare it with the costs to the attacker.

The earliest failure models like [34] consider one class of failures only, mainly crashes and Byzantine failures. Hybrid failure models, on the other hand, discern several types of failures. There are a number of them in the literature which differ primarily in the number of classes of process failures considered: Early models like [69] distinguish just manifest and Byzantine failures, whereas fully-fledged models like the ones of [71] and [5] provide a reasonably complete classification of all conceivable failure modes. Hybrid failure models are interesting, in particular for small-sized systems, since they allow to exploit the fact that less severe failures can usually be handled with fewer processes than more severe ones. However, none of the existing hybrid failure models that are applicable to consensus algorithms also covers link failures explicitly.

In fact, there are only a few failure models for synchronous systems in the literature that deal with link failures at all. One obvious way is to attribute link failures to (sender-)process failures, as in [26], for example. Still, allowing at most f_ℓ link failures in every process, as in our model, would generate many failure patterns where all $f = n$ processes must be considered faulty; Figure 1.2 shows an example for $n = 4$ and $f_\ell = 1$. This (falsely) suggests that problems like clock synchronization and consensus are not solvable in presence of link failures.

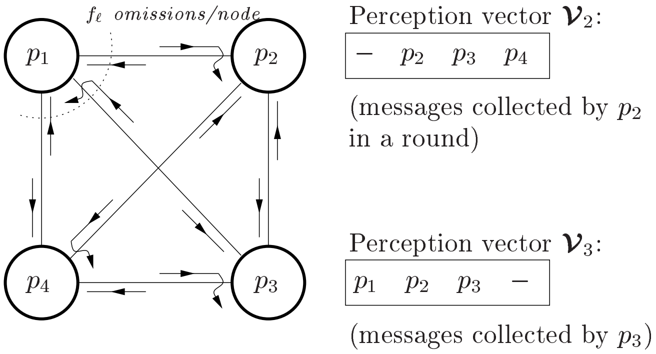

Figure 1.2. *Example of a 4-process system with $f_\ell = 1$ receive failures per process in each round, where all processes must be considered faulty in traditional process-centric failure models.*

There are a few papers [46, 52, 64] on distributed consensus that incorporate a limited number of link failures explicitly. They all assume, however, that only $f_\ell = \mathcal{O}(n)$ of links may be faulty in the whole system. In sharp contrast, our results will reveal that Byzantine agreement can even be solved when $\mathcal{O}(mn^2)$ link failures occur in the system during m rounds of execution. Byzantine agreement algorithms can hence tolerate about the same number $\mathcal{O}(n^2)$ of link failures as leader election algorithms [1, 63], without, however, sacrificing the ability to deal with Byzantine faulty processes. For example, our algorithm ZA analyzed in Section 5.5.2 would easily achieve consensus among all processes in the example of Figure 1.2.

1.3 Outline

In Chapter 2, we will give an introduction to security and will describe some of the mechanisms required to achieve security goals. We will explain symmetric and asymmetric encryption, hash functions, signatures, and certificates, since we will make use of these mechanisms in our authentication protocols.

Chapter 3 is concerned with the hardware support we require to achieve authentication. In order to filter out potential denial-of-service attack messages as soon as possible, we have incorporated several message filters into our hardware. Additionally, our hardware offers support for certificate management, for encryption/decryption, and for hashing.

Chapter 4 presents our protocols related to authentication. Our authentication protocols pay particular attention to denial-of-service attacks and are based on a combination of client puzzles [32] together with signatures used during the initiation of a (streamed) communication and message authentication codes [70] employed in subsequent messages. We will present protocols for obtaining the public key of a node from the certification authority, for changing the public key of a node, and for authentication itself.

Chapter 5 is dedicated to authenticated consensus. We will define a hybrid failure model which incorporates link failures and is hence well suited for wireless networks. We will also derive some lower bounds and impossibility results concerning consensus under link failures. The main part of the chapter is devoted to the analysis of several Byzantine agreement algorithms [34] under our hybrid failure model. We will see that they are all applicable in our failure model, which will re-confirm that authentication can significantly reduce the number of nodes required to achieve agreement.

Chapter 6 contains the conclusions and suggestions for future work.

Chapter 2

Security Methods

Over the last few decades, computers and embedded systems have been assigned an ever-increasing importance in our lives, providing us with services we have grown accustomed to. We send e-mails to our friends and co-workers, order supplies through the Internet, or withdraw money from an ATM. As long as these services are available, we do not particularly care how they are implemented and whether there might be some risks involved. We do care, however, if our e-mails are intercepted and their contents are misused, if we get a bill for something we have never ordered and have no means of proving that, or if our bank account is highly overdrawn because someone else has gained access to it. This is the point where we start demanding actions to prevent such outrageous occurrences in the future, and unfortunately, this has also long been the point where security has entered the picture for the first time. Even now, after many years of adverse experiences, security often is still only an afterthought when setting up a company network or web server. Even e-commerce so far frequently suffers from a lack of adequate security, just think of the still too many company websites which ask for credit card information without providing security against misuse, and a lot of effort is expended by the security community to design protocols which provide sufficient security to both customer and salesperson. Basically, security helps to guarantee that services are behaving according to their specifications by ensuring that no unauthorized access can disrupt the protocol. In the above examples, security would encrypt the e-mail to prevent the eavesdropper from reading it and would ensure that an online order cannot be associated with a wrong client.

Security can be associated with different goals, which are roughly subsumed under the following terms:

Accounting logs service requests of users. The goals are manifold, ranging from the ability to trace back system attacks up to the maintenance of disk quotas for each client. The service depends on authentication.

Authentication aims to guarantee the identity of principals during communication. In *one-way* authentication, only one of the partners is sure about the identity of the other, in *two-way* authentication both peers know with whom they are talking.

Authorization strives to grant access only to special groups. The requirement depends on authentication to ensure who the principal is, but additionally has to check whether the principal is allowed to access given services.

Integrity guarantees that message modifications are either impossible or can at least be detected by the receiver.

Nonrepudiation prohibits a sender from later denying that it has sent a message.

Secrecy tries to keep the contents of a communication exchange hidden from possible spies. There are different degrees of secrecy, from simple encryption of the message's contents over hiding the message's sender and recipients up to hiding communication at all.

These goals are typically achieved by means of one or more security algorithms. In general, the less complex the algorithms and their interaction, the less chance there is for failure to achieve the goals. It is immediately apparent that authentication plays an important role in attaining most security goals because it links actions to users.

Unfortunately, the need for security implies that there are parties (called *attackers* or *intruders*) who want to prevent us from reaching these goals. Intruders typically try to attack the security algorithms to prevent them from attaining their goals. Furthermore, they may also try to cause the protocol to fail. This type of attack, called *denial-of-service* (DoS), tries to cause a server to deny its service to a legitimate user. A typical way to achieve this goal is to exhaust the server's resources with bogus service requests. A more subtle method would be to replace the message from a legitimate user with a fake message that causes the server to abort the protocol run, or to send additional messages with this result. Originally, denial-of-service attacks were not considered in security protocols, possibly because they are very hard to prevent, especially if the intruder is assumed to have full control over the network. Due to attacks on popular Internet servers like Yahoo or Amazon and the well-known SYN attack on the TCP protocol, the topic now receives more attention and counter-measures like cookies (see [48] for an overview) or client puzzles [32] have been proposed.

Since security protocols generally need techniques like encryption, hashing, or signatures to achieve their goals, we will give a brief overview of these methods. A comprehensive treatment can be found e.g. in [60].

2.1 Encryption

Encryption schemes are parted into *symmetric* and *asymmetric* ones. *Symmetric methods* use one key for encryption and decryption. In consequence, both the sender and the receiver have to know the key. This is why such methods are often called *shared key systems*. Of course, the receiver must obtain the key prior to decrypting the message, and transmission of the key must be secure for the scheme to work. If the receiver compromises the key, the sender must change its key as well. This makes symmetric key schemes more problematic than asymmetric ones.

Examples of symmetric systems are

DES (Data Encryption Standard): This method encrypts blocks of 64 bits with a key of length 56 (the last 8 bits of the block are a checksum). Details of the algorithm can be found in [60]. The algorithm is quite simple and can hence be easily implemented in hardware, which makes it very fast. Throughput of 1 Gbps is possible.

Until recently, DES fell under the export restrictions of the USA, preventing longer keys to be used. However, the 56-bit version of DES has already been cracked using a brute-force algorithm which checks all possible keys until it finds the right one. The effort necessary to launch such an attack is relatively moderate, so 56-bit DES cannot be considered to be secure. A key length of 128 bit is therefore recommended by the security community. Nevertheless, DES is a problem because it is feared that the NSA (National Security Agency) of the USA, which has been involved in the development of DES, may have left some backdoor to decrypt messages even without the key.

Triple DES: This is a variant of DES which encrypts the data with a 56-bit key, decrypts it using another 56-bit key, and then encrypts it again. This has more or less the same effect as a longer key, making a brute-force attack unfeasible. But of course it slows down the algorithm.

IDEA (International Data Encryption Algorithm): The method has been developed at the ETH Zurich (Switzerland) [33] and also encrypts data blocks of 64 bits length, but uses a 128-bit key. It avoids several drawbacks of DES, and the long key makes brute-force attacks unfeasible. A hardware implementation done in 1994 [81] yielded a throughput of over 177 MBit/s. The algorithm is considered to be very secure, but is protected by a patent.

Rijndael: The Rijndael algorithm [12] has been developed quite recently and has been selected as the new Advanced Encryption Standard (AES) [17] by the American NIST (National Institute of Standards and Technology). The algorithm along with several other candidates has undergone intense public scrutiny. It can use keys of 128, 192, and 256 bits to encrypt data of 128 bits per block. The algorithm replaces DES as the standard. Its selection criteria included that the algorithm should be faster than Triple DES, be freely available and have been publicly scrutinized. Further information can be obtained from the AES homepage at *http://csrc.nist.gov/encryption/aes/*.

The above methods are all *block ciphers*, that is, they encrypt the plaintext one block at a time. However, this leads to the problem that an intruder can replace blocks and thus manipulate the message without being detected. For this reason, *cipher block* *chaining (CBC)* has been introduced. In this method, the plaintext of a block is XOR-ed to the encrypted previous block before being encrypted itself. The first block is encrypted random data to make every ciphertext unique.

Asymmetric methods use a pair of keys (K_{pub}, K_{priv}) which are related to each other. Anybody may obtain the public key K_{pub}, but only the owner of the pair knows the private key K_{priv}. The keys are chosen such that encrypting a message m first with one and then the other key is commutative and results in m, i.e., $K_{pub} \circ K_{priv}(m) = K_{priv} \circ K_{pub}(m) = m$. Knowing the public key does not enable anyone to guess the private key, and there is only one private key which is the inverse of the public key.

In the remainder of the thesis, we will denote K_p as the public key of user p, and K_p^{-1} as p's private key. Encryption of a message m is denoted by $\{\mathrm{m}\}_K$, where K is either the public or the private key.

Asymmetric schemes enable a user Bob who knows K_A to send an encrypted message to Alice, the owner of K_A^{-1}. Since the private key is only known to Alice, it is guaranteed that nobody else can read the message. On the other hand, Alice may encrypt a message using K_A^{-1} and send it to Bob who applies K_A to obtain the plaintext. Since only K_A^{-1} is the inverse of K_A and only Alice knows K_A^{-1}, Bob is assured that the message has been sent by Alice. Thus, asymmetric methods can be used to electronically sign messages. Their major disadvantage, however, is that due to the difficult computations involved, encryption takes much longer than with symmetric schemes.

Well-known asymmetric schemes are

RSA: This method, which has been named after its inventors Rivest, Shamir, Adleman [50], has been the first asymmetric method to become widely popular. It is based on the hardness of factorization of large numbers into their primes. The problem of computing from a given n the two prime numbers p, q for which hold $n = p * q$ is extremely hard for large n.

ElGamal: This scheme, also named after its inventor [23], uses discrete logarithms. The problem of computing from a given y, g, and p the x for which holds $y \equiv g^x \ mod \ p$ is extremely hard for large p (at least as hard as the factorization problem utilized by RSA).

The main disadvantages of asymmetric schemes are that the execution of the algorithm is very time-consuming and that secure keys are much longer than in symmetric schemes (e.g., RSA needs at least 1024 bits). Additionally, it is hard to change the public key, since most users will download it only once and then store it locally, although this is not a problem in W2F since we do not consider human users. On the positive side, the schemes are scalable because the effort necessary to break the method grows exponentially with the size of the chosen numbers, so whenever computers get fast enough to break a key, one can always use larger numbers to make brute-force attacks unfeasible again.

2.2 Hash Functions

In contrast to encryption methods, which aim to keep the contents of messages secret, *hash functions*, which are also called *message digest functions*, try to guarantee the integrity of messages. The idea is to append to the message a short sequence of bytes which has been computed from the message itself. The function used for obtaining the hash value should be easy to compute, result in a small hash value of fixed length independent of the original message length, and it should be impossible to construct a message from a given hash value. In fact, it is often demanded that it should be unfeasible to find two different messages that have the same hash value, i.e., the hash function should be *collision-free*.

The sender of a message computes its hash, sends the message to the receiver and uses some secure way to transmit the hash of the message to the receiver. The receiver also computes the hash of the message and compares it to the hash stated by the sender. If the two values match, then the message has not been tampered with.

MD5: *Message Digest 5* [49] is the last of a series of message digest functions developed by Ron Rivest (the others are MD2 and MD4). It computes hash values of 128 bits length, has been widely used, is very fast, but unfortunately has some weaknesses that allow to find collisions. Therefore, its use is not recommended.

SHA: The *Secure Hash Algorithm* [18] has been developed by the NSA and is considered to be one of the best currently known methods. It computes hash lengths of 160 bits and operates on blocks of 512 bits.

RIPEMD-160: The algorithm *RIPEMD-160* [15] is a strengthened version of RIPE-MD. RIPE-MD has evolved from an EU project and is very similar to SHA, but has a hash value length of only 128 bits. Since a longer hash value makes the system more secure against brute-force attacks which simply try out all possible combinations until they find a collision, RIPEMD-160 results in a 160 bit hash value like SHA. It also operates on data blocks of 512 bit size.

Currently, SHA and RIPEMD-160 are the best hash functions available. Their performance is comparable, although SHA is slightly faster than RIPEMD-160 [8]. None is protected by any patents.

Hash functions employing shared secret keys are called *message authentication codes* (MACs) [70]. Since the resulting hash value is based both on the message and on a secret key known only to the sender and the receiver, it can be sent with the message. Even if an intruder intercepts the message, he cannot replace the hash without knowing the key and therefore cannot forge the message. Obviously, MACs guarantee both the integrity of the message as well as the authenticity of the sender, provided that the secret key is only shared between the sender and the receiver and that the receiver is honest.

2.3 Signatures

As we have seen in the previous section, hash functions guarantee the integrity of the message, provided that the hash value is transmitted to the receiver in a secure way. A signature normally does a lot more: It binds the sender (who has signed a message) to the message. This involves guaranteeing the authenticity of the sender, the integrity of the message, and the connection between the message and its signature. Like a MAC, a signature can simply be transmitted with the message, because it is guaranteed that only the sender can generate a valid signature for a message.

Fortunately, there is a simple method to achieve all these requirements for electronic signatures, which is based on asymmetric key encryption in combination with message digest functions: The sender Alice simply computes the hash value of the message, encrypts this value with her private key, and appends the hash to the plaintext message. The receiver Bob decrypts the hash and compares it with his own computation of the hash of the message. This guarantees Bob that the message has not been tampered with (the hash values are equal), that Alice has been the sender of the message (only Alice could encrypt the hash value with her private key), and that Alice has sent this particular message (Alice has sent the correct hash value). In order to guard herself against possible attacks or misuse of her message (e.g., sending the same bank transaction request twice), Alice should also include a timestamp or some other unique value into her documents before she signs them. If the message should also be secret, Alice first appends her signature to the plaintext message and then encrypts the complete message (including her signature) with Bob's public key.

2.4 Certificates

The existence of asymmetric encryption schemes gave rise to the need for secure publishing of the public keys. After all, we cannot simply accept just any public key sent to us by a node. We must be certain that the key sent to us really does belong to the node, so there must be some way to securely bind a node identifier to a public key. The general solution to this problem is to use a *public key certificate* which is issued by a *certification authority* (CA). The CA is a trusted authority whose signed messages are accepted as correct. Ideally, it should be a fault-tolerant and intrusion-tolerant service like COCA [80]. It is vitally important for the security of all algorithms relying on public keys that the CA and its key are never compromised.

The CA issues certificates, which are simply signed messages binding the name of a node to its current public key. Naturally, nodes may change their keys from time to time, so the CA will in time issue several certificates for a node. Old keys are not secure anymore, hence they and their certificates must be invalidated somehow. This is typically done with *key revocation lists* which contain all revoked keys of a node.

The X.509 certificate standard [30] is very popular since it is employed in the SSL protocol used in the Internet. The webpage *http://java.sun.com/j2se/1.4/docs/guide/security/cert3.html* gives a short overview. An X.509 certificate contains numerous information like the protocol version number or the signature algorithm. For the purposes of this thesis, we will simply assume that a certificate consists of the name and public key of a node p. In order to recognize outdated certificates, we require that the certificate for any node p includes a version number ν_p with the property that $\nu_p < \nu'_p \Leftrightarrow$ the certificate containing ν_p is older than the certificate containing ν'_p. Hence, we do not need key revocation lists since we can always recognize a new certificate by its higher version number.

Chapter 3

Hardware Support

The W2F security system is parted into a hardware and a software layer. The hardware layer is located between the network controller and the physical medium and can change outgoing messages immediately before they are sent and filter incoming messages before they are passed on to the software. The software layer is directly above the message event handler and can also filter any incoming message.

Since the W2F system is an ad-hoc network with sparse connectivity, denial-of-service attacks are a lot easier than in the normally assumed static (point-to-point) networks. Therefore, we decided to include support against such attacks in the hardware layer of our security system. Unfortunately, we cannot prevent attackers from sending messages on our network, although we will try to make unauthorized access to the W2F network as hard as possible. Thus, we cannot prevent nodes from receiving unwelcome messages, and we have no control on the network load, which is a denial-of-service attack opportunity in itself. We will of course try to locate and exclude nodes generating excessive network load, but until then, a node is open to all kinds of messages sent by intruders and should recognize fake messages as soon as possible. To this aim, we propose several hardware mechanisms that help to detect and filter out invalid messages.

Hardware support is generally very important in our W2F project and it is realized with a so-called *W2F box* attached to every node. The box handles network access and contains all additional hardware features we require. It is a trusted computing base and will most likely use a secure co-processor [65]. At the least, we require some non-volatile storage space which does survive a node crash and have to assume that the box does not disclose any secrets (namely, the secret keys of the node). If the secrets of the box are disclosed, the owner node is regarded as compromised. Corrupting the hardware over the network must be prevented, so an attacker actually has to gain physical possession of the box to corrupt it. Since we envision W2F to be a fieldbus which is limited to the confines of a factory or some other restricted area, this already reduces the number of attackers able to gain access to a box. Nevertheless, the final W2F box will most likely feature a secure co-processor with tamper-proof or at least tamper-aware memory. Initially, the box knows the public/private key pair of the node, the public key of the certification authority, and two public values n and g necessary for Diffie-Hellman style key agreement, see Section 4.4.

Our W2F box will include support for message timestamping and for security related operations and will be capable of filtering out messages according to several criteria. In the remainder of this chapter, we will describe these features.

3.1 Timestamping

In W2F, we will have clock synchronization available (see [61] for an excellent overview on interval-based clock synchronization in fully connected systems). However, since we have an ad-hoc network where the degree of connectivity is not guaranteed, clock synchronization depends on authentication to minimize the capacity of Byzantine faulty nodes. In consequence, we will assume that a timestamp T taken at real-time t contains the local clock time $C(t)$ and a flag stating whether the time is valid or not. If the timestamps T_p and T_q of two nodes p, q are both valid and if the nodes are both correct, then $|C_p(t) - C_q(t)| < \pi$ where π is the global precision of the clock synchronization algorithm. If the flags are not valid, then the timestamps are arbitrary and bear no relationship. Every node is in this state immediately after joining the W2F network for the first time, and a node may fall back into the unsynchronized state if the synchronization of the clock cannot be guaranteed (most likely due to sparse network connectivity). Apart from the global precision π which is valid for all nodes, clock synchronization can also guarantee a precision $\pi_S \leq \pi$ within a synchronous subsystem S if the connectivity of the subsystem is sufficient. In this thesis, however, we will only use the global precision π.

Specification 1 (Timestamping) *Messages are timestamped automatically by hardware at the sender and the receiver. Care is taken to insert the timestamps as close to the physical transmission as possible to exclude host latencies. The timestamps are inserted into reserved fields of the message on-the-fly, i.e., during the message transmission/reception operation on the physical layer.*

This hardware mechanism has already been successfully used in our project SynUTC, which was dedicated to high-accuracy clock synchronization [56].

3.2 Hashing and MACs

The W2F box is able to compute a hash on-the-fly. We propose to use RIPEMD-160 or an equally secure algorithm that is fast enough. According to Bosselaers et al. [8], RIPEMD-160 achieves a hashing speed of 19.3 Mbit/s as a portable C implementation on a 90 MHz Pentium. We assume a W2F data transmission speed of 1 Mbit/s, so any hash algorithm which achieves at least 1 Mbit/s in the hardware implementation can be used for hashing messages on-the-fly. Since the hash function is also used to generate message authentication codes (MACs), it should in fact be slightly faster than 1 Mbit/s. If we use the mechanism described in [60, p. 459], where at least 64 bits of the key are appended to every message block, and if we assume a message block size of 512 bit as used by RIPEMD-160 and SHA, then the hashing function must have a throughput of at least (64+512)/512 = 1.125 Mbit/s, which is easily achieved by RIPEMD-160.

Specification 2 (Hashing) *The W2F box can on-the-fly compute the hash value of a message that is being transmitted or being received.*

Specification 3 (MACs) *The W2F box can on-the-fly compute a keyed one-way hash value. The key is shared between sender and receiver.*

We will use the notation $\sigma_p^h(m)$ to denote that node p has included a MAC for message m.

3.3 Signatures and Encryption

The W2F box can sign messages using a public-key signature scheme. To achieve this, we assume that there is a (hardware-)programmable table of public keys with entries of the form $\langle p, K_p, \nu_p \rangle$ from where the hardware can take the public key K_p with version number ν_p of node p during signature verification. We also assume that the hardware is capable of recognizing new key certificates signed by the certification authority and can update the table accordingly. The hardware takes over a new certificate $\mathrm{Cert}_p(K'_p)$ consisting of the key K'_p and the version number ν'_p for a node p if it has currently stored the data from a certificate $\mathrm{Cert}_p(K_p)$ and if the new version number fulfills $\nu'_p > \nu_p$.

We do not commit ourselves to any specific public key encryption method, we just require that the keys should be reasonably hard to find out. We recommend using a scheme where both signature computation and verification are fast (at least when implemented in hardware), even if it means that the scheme is less secure and we have to change keys more often. We place, however, more weight on the signature computation speed than on the verification speed.

Specification 4 (Signature) *We assume that our W2F hardware can sign messages (almost) "on-the-fly" during the send operation and that it is able to verify signatures of incoming messages "on-the-fly" during the receive operation if a fresh, i.e., a currently used, public key of the sender is available to the hardware. The send timestamp must be included in the signature, so precomputation and caching of signatures is not possible if hardware timestamping is turned on. A signature for a message can also be requested by the software, in which case the signature is returned to it.*

We will use the notation $\sigma_p(m)$ to indicate that node p has signed the message m.

Note that the signature mechanism is not as on-the-fly as we made it sound. Computing and verifying signatures are costly operations and can even take seconds if performed in software [60, p. 488, Table 20.2]. However, we trust that hardware implementation will speed up computation and verification time considerably, so we expect them to be at most in the 1-10 ms range, probably even lower for a fast and less secure algorithm than the DSA algorithm measured in [60]. Secondly, we can measure the signature computation time by inserting a second timestamp at the end of the message immediately before signature computation is completed. This allows the receiver to exactly measure the signature computation delay, which is then simply added to the transmission delays the receiver attributes to the message. Verification also takes time, but since it is spent at the receiver, it does not influence the filtering mechanism. It is, however, a weak point for a DoS attack. Nevertheless, our emphasis is on the speed of the signature, it should be as fast as possible to allow the intermediate nodes of a connection to sign a message without excessively delaying it. Measurements by Wiener [76] suggest that verification is fast in RSA (0.6 ms on a 200 MHz Pentium Pro), whereas signing is slow (43 ms). In contrast, algorithm DSA, which is used in the Digital Signature Standard DSS [16], signs in about 7 ms and verifies in 27 ms.

Specification 5 (Encryption) *The W2F box can encrypt an out-going message with the public key of the receiver if a fresh public key is available. The software can also request that a particular message is encrypted with a given key, in which case the resulting message is returned to it.*

We will use the notation $\{\mathrm{m}\}_{K_q}$ to indicate that a message has been encrypted with the public key of the receiver node q.

3.4 Message Filters

The W2F box contains the following hardware message filters, which can be selected independently of each other on a per-node basis. If they are active, then they assume that their respective hardware mechanisms which we described above are turned on as well. Messages pass through the filters in the following order:

Receiver Filter: The W2F hardware is capable of filtering out messages not addressed to the receiver. This is a well-established feature of network controllers. The filter uses a simple comparison of the receiver's name with the receiver address field of the message and discards the message if there is no match.

Sender Filter: The hardware can also filter out messages from unwelcome senders. Such a feature is useful for filtering out messages from known intruders. The list of unwelcome senders must be accessible to the hardware. Every entry in the list is compared to the sender address field and the message is discarded as soon as a match is found.

MAC Filter: If MAC filtering is active, then a message is only passed on to the software if the MAC could be verified. If there is no MAC key stored for the sender, then the message is discarded at once, otherwise it is discarded if the computed MAC does not match the one appended to the message by the sender.

Timestamp Filter: In a synchronous subsystem where message delays are known, the W2F hardware can filter out messages which were excessively delayed. If we assume that we know the message transmission delay interval $d_{pq} = [d^-_{pq}, d^+_{pq}]$ from sender p to receiver q, and if we know the upper and lower bounds $\rho^\pm_q$ on the clock drift of receiver q, then if q's timestamps are valid, a message from p containing a valid send timestamp T_p will only be passed on to the software if $C_q(t_r) \in [C_p(t_s) - \pi + d^-_{pq}\rho^-_q, C_p(t_s) + \pi + d^+_{pq}\rho^+_q]$, where $C_q(t_r)$ is the receive timestamp of node q, $C_p(t_s)$ is the send timestamp of p, and π is the precision of the clock synchronization algorithm. Of course, this feature can only be used if the transmission delay bounds are known. Periodical transmission delay measurements are required for clock synchronization, so we can assume that we are given transmission delay bounds for the actual route taken. If for some reason we do not have the delay information, or if either the send or the receive timestamp is invalid, then messages are simply passed through.

Signature Filter: If signature filtering is active, then a message is only passed on to the software if the signature could be verified. If no fresh public key is available for checking the signature, then the message is passed on unverified, but steps are taken to obtain a fresh key. The recipient can then request verification of the message at some later time.

Only messages which pass through the first four filter steps will actually require a time-consuming signature verification. The filters are ordered according to increasing resource consumptions at the receiver. An intruder can easily circumvent the address filters and will not have much problems with the timestamp filter, but the MAC and signature filters are very hard (virtually impossible) to bypass by intruders.

The timestamp filter is only useful as long as timestamps are valid. If MAC filtering is active, then timestamp manipulations are recognized before the timestamp is checked. If signature filtering is active, then an intruder launching a DoS attack against the computationally expensive signature verification can circumvent the timestamp filter easily by setting the valid flag of the send timestamp to false. Nevertheless, the filter is useful for preventing accidental replays and should be used even if MAC filtering is not possible.

Chapter 4

Authentication

Authentication is one of the core functionalities of W$_2$F, since it allows various algorithms like consensus or clock synchronization to tolerate more failures and work in a less connected environment. For example, there is a symbiotic relationship between authentication and clock synchronization: Our authentication algorithms can live without clock synchronization and thus without the hardware timestamp filter, but if it is available then denial-of-service attacks are severely restricted and can be filtered out by hardware most of the time. Clock synchronization, on the other hand, requires $2f + 1$ disjoint connections between nodes without authentication and just $f + 1$ such connections if authentication is available, where f is the number of faulty nodes and links in the system. So the proper startup sequence will be to first establish authentication without synchronized clocks, thereby perhaps even just allowing clock synchronization to kick in, and then use the synchronized clocks to keep attackers at bay. Obviously, the system is most vulnerable to DoS attacks during the phases where clock synchronization is not available.

Many authentication protocols are available and are used in existing systems, like the CCITT X.509 standard [31], Kerberos [44], or the Transport Layer Security protocol TLS [13], the successor of the widely used SSL-protocol. Originally, we had planned to use an existing protocol that has already been extensively analyzed, since security protocols tend to contain subtle flaws, as many papers analyzing such algorithms like [36, 22, 6, 62] (to name only a few) show. However, we found that the system assumptions of most algorithms do not fit well to our ad-hoc network which has a high probability of mobile intruders launching denial-of-service attacks. For this reason, we decided to develop own protocols which integrate denial-of-service prevention techniques. Our own protocols are of course neither message efficient nor round efficient, see [25] for optimal algorithms in that respect, but are designed to minimize the feasibility of denial-of-service attacks.

We will use the well-known Alice-and-Bob notation for presenting the algorithms in this chapter. We will only give an informal definition of this notation, please see [39] for a more formal one.

Definition 1 *An* Alice-and-Bob notation *is a numbered sequence of messages of the form A → B: M. The notation A → B: M signifies that Alice sends a message with the contents M to Bob. In our case, a message with number $k > 1$ is sent only after the message with number $k - 1$ has been received. A ′ after a sequence number denotes an alternative message which may be sent instead of the original one.*

In the messages, we will use the following terms:

A, B, CA: identifier of Alice, Bob, or the certification authority

Data_X: data included by node X

N_X, N'_X: nonces (i.e., unique identifiers) generated by node X

S_X: sequence number computed by node X

T^s_X: send timestamp of node X

T^r_X: receive timestamp of node X

`text`: message type

As an example for a well-established authentication protocol using public keys, we demonstrate the Needham-Schroeder protocol [42], which requires a trusted authority for the certificates. The original protocol had a flaw which has been pointed out by Lowe [36]. Another flaw has been found by Meadows [40]. The following version already incorporates the fix proposed by Lowe, which also repairs the flaw found by Meadows.

Algorithm 1 (Needham-Schroeder Authentication [42])

1. A → CA: B
2. CA → A: $\{\mathrm{K}_B, \mathrm{B}\}_{K_{CA}^{-1}}$
3. A → B: $\{\mathrm{N}_A, \mathrm{A}\}_{K_B}$
4. B → CA: A
5. CA → B: $\{\mathrm{K}_A, \mathrm{A}\}_{K_{CA}^{-1}}$
6. B → A: $\{\mathrm{N}_A, \mathrm{N}_B, \mathrm{B}\}_{K_A}$
7. A → B: $\{\mathrm{N}_B\}_{K_B}$

In the protocol, Alice requests Bob's public key from the certification authority (CA) in message 1 and receives the certificate in message 2. She then sends a nonce N_A and her name to Bob, who in turn obtains Alice's public key in messages 4-5. Bob replies with Alice's and his own nonce N_B as well as his name in message 6, and Alice completes the protocol by returning Bob's nonce in message 7. After the run, both parties are certain about the identity of their peer.

Messages 1-2 and 4-5 are just queries for the current certificate of the peer and can in fact be decoupled from the actual authentication, leaving a three-message authentication protocol between the peers.

1. A → B: $\{\mathrm{N}_A, \mathrm{A}\}_{K_B}$

2. B $\rightarrow$ A: $\{N_A, N_B, B\}_{K_A}$
3. A $\rightarrow$ B: $\{N_B\}_{K_B}$

However, we note that the protocol is extremely vulnerable to denial-of-service attacks since it employs computationally expensive encryption operations in all three messages. An intruder just has to replay Alice's (and other nodes') first messages to keep Bob busy decrypting the messages and encrypting the replies. The nonces do not help Bob to recognize replays, since he can hardly store all nonces ever sent by Alice. Similar problems occur in other authentication protocols like Kerberos [44], CCITT X.509 [31], the Neuman-Stubblebine protocol [43] and many more. The problem is alleviated in protocols like Kerberos and X.509 which rely on clock synchronization, because the receiver can at least check the validity of the timestamp included in the message and will not reply to old messages. But he must still decrypt every message that is received. For this reason, we have opted for using a combination of client puzzles [32, 4] and message authentication codes (MACs) [70].

In the remainder of this chapter, we will first explain how we integrate client puzzles into our authentication protocols, and then present our algorithms for authentication. In general, authentication using a public key cryptosystem must deal with three problems:

Key Distribution: How do nodes get the public key?

Authentication: How do nodes find out who they are talking to?

Key Revocation: How is the key of a node changed?

The Needham-Schroeder algorithm gives a combined solution for the first two problems. The third problem arises because given enough time, an intruder can find out private keys by using brute-force methods, forcing all nodes to change their keys from time to time. We feel that one should treat these issues separately and allow the designer to schedule the algorithms as necessary. When designing our algorithms, we paid attention to minimize the potential for DoS attacks.

For our algorithms, we will assume a synchronous (sub)system between Alice, Bob, and the CA where nodes are connected by at least one correct path. A *correct path* is either a direct connection or a relayed connection with only non-faulty relay nodes and links. We feel that this strong assumption is justified by the fact that the CA is distributed and its servers are spread over the whole network, hence there will be CA servers in the vicinity of Alice and Bob, allowing synchronous communication. The amount and frequency of communication between Alice and Bob, on the other hand, will most likely be inversely proportional to the distance between the nodes, since it makes sense that nodes working closely together should also physically be close together. So most of the time, Alice and Bob will have a synchronous communication as well. But we will of course consider violations of these assumptions in Sections 4.2 and 4.5.

4.1 Client Puzzles

Basically, we concur with the opinion stated in [4] and [38] that the attacker should be the first to commit its resources, which for our purposes consist of memory, computation time and signature verification time. The client puzzle approach of [4] is particularly attractive since it places practically all load on the client. The method is based on asking the client to solve the following equation by brute force:

$$h(C, N_S, N_C, X) \quad = \quad \underbrace{000\ldots000}_{k}Y$$

Here, h is a one-way hash function like SHA, C is the identity of the client, N_S is a nonce obtained from the challenge of the server, N_C is a nonce generated by the client for this puzzle, and X is the solution of the puzzle. The task is to find X so that the resulting hash value has k leading zeros. The remaining bits Y are not important. The difficulty of the puzzle is controlled by k, ranging from easy for small k up to impossible if k encompasses all bits of the hash.

We present the algorithm of [4] in the Alice-and-Bob notation and let Bob act as the server: Periodically, Bob precomputes and signs a message $\sigma_B(T_B, k, N_B)$ consisting of a timestamp T_B, a nonce N_B, and the difficulty parameter k. This message serves as the puzzle. When Alice contacts Bob, the following exchange takes place.

Algorithm 2 (Client Puzzle Authentication [4])

1. A $\rightarrow$ B: Request
2. B $\rightarrow$ A: $\sigma_B(\mathrm{T}_B$, k, $\mathrm{N}_B)$
3. A $\rightarrow$ B: σ_A(B, A, N_B, N_A, X)
4. B $\rightarrow$ A: σ_B(A, B, N_A)

With the first message, Alice sends her communication request to Bob, who replies with his puzzle $\sigma_B(T_B, k, N_B)$ in the second message. In the third message, Alice sends the nonce N_A and her solution X to the corresponding puzzle $h(A, N_B, N_A, X)$ = 000. . . 000Y. After receiving this message, Bob makes certain that the triple A, N_B, N_A has not been used before and verifies the solution X. After that, he may commit his own resources to the connection and will store A, N_B, N_A as long as N_B is fresh. He also verifies the signature σ_A to authenticate Alice. If Alice needs to authenticate Bob as well, then Bob should send the last message.

Using this method would be a good choice for authentication except for three drawbacks: First, it at least triples the number of messages required, since every message from Alice to Bob is challenged, and only after the reply does Bob react to the contents of the message. If Bob broadcasts his challenge from time to time, then there are only two messages (three if Alice needs to authenticate Bob) involved, but the network load in our system would be even higher due to the broadcast. Secondly, the protocol has only been designed to protect servers. Alice must always authenticate messages from

Bob and is thus vulnerable to DoS attacks while waiting for messages 2 and 4. Although we can reduce the vulnerability until message 2 by requiring that Alice includes a nonce in her first message, which should be sent back by Bob, an attacker knowing the nonces can still start DoS attacks. Third, the method requires timestamps or some other freshness indicator. We cannot guarantee that clock synchronization is always available. Therefore, we do not want to use this kind of authentication for the whole time. On the other hand, if we do not use something that is easily checked, we will end up checking fake signatures all the time. Hence, we will use the following combination of techniques for authentication:

If Alice has never talked to Bob before or if there has been a period of silence which was long enough for Bob to remove all communication data stored for Alice, then we use the client puzzle approach. But when sending the puzzle solution, Alice includes a secret key encrypted with Bob's public key which is intended for MAC computation. After receiving the message and checking the correctness of the solution, Bob finally commits his own resources by storing Alice's public key and verifying the signature. Bob also decrypts and stores Alice's MAC key. The fourth message does already include a MAC and serves as an acknowledgment that Bob has received the key. From then on, Bob turns on MAC filtering for Alice. The hardware can even recognize this state automatically by the fact that a shared MAC key is stored for Alice. All further communication with Alice is now authenticated with MACs. Since shared keys are normally considered to be relatively insecure, Alice should change the key from time to time. If Bob does not get a new key within a certain time, he will fall back to the initial phase and send Alice a client puzzle again.

The general authentication protocol has the following format (items in brackets are inserted automatically by the hardware):

Algorithm 3 (General Authentication) *Our general authentication protocol has the following format:*

1. A $\rightarrow$ B: [B, A, T^s_A, T^r_B], N_A, `puzzle request`
2. B $\rightarrow$ A: [A, B, T^s_B, T^r_A], N_A, $\sigma_B(T_B, k, N_B, S_B)$
3. A $\rightarrow$ B: σ_A([B, A, T'^s_A, T'^r_B], N_B, N'_A, X, $Data_A$)
4. B $\rightarrow$ A: σ_B([A, B, T'^s_B, T'^r_A], N'_A, $Data_B$)

3′. A $\rightarrow$ B: σ_A([B, A, T'^s_A, T'^r_B], N_B, N'_A, X, $\{K_{AB}\}_{K_B}$, $Data_A$)

4′. B $\rightarrow$ A: σ^h_B([A, B, T'^s_B, T'^r_A], N'_A, $Data_B$)

In addition to the nonce sent by Alice in the first message, we also include a sequence number S_B with the puzzle message $\sigma_B(T_B, k, N_B, S_B)$. It serves as a crude freshness indicator in case the clock synchronization fails. Although it will not prevent Alice from accepting an outdated puzzle, she will at least recognize a more recent puzzle and can then discard the old one. The *Data* items simply contain node data. Messages (3′) and (4′) depict the communication setup for a longer-lasting communication. Here, Alice sends an encrypted MAC key used for MAC filtering in all subsequent communication, see Section 4.2 for details.

As we have indicated, an intruder can launch DoS attacks against Alice since she must verify the signature of every message allegedly sent by Bob which passes through the timestamp filter. We have explained in Section 3.4 that this filter can easily be circumvented if the intruder marks the send timestamp as invalid. But as long as clock synchronization is available, we can make sure that an intruder cannot cause Alice to accept an outdated puzzle: If we assume that Bob creates puzzles with period P_{puzzle}, then we have two consecutive puzzles with times T_B and $T_B + P_{puzzle}$. If Bob sends a fresh puzzle with time T_B to Alice, then Alice will receive and must accept this puzzle at her local time T_A^r somewhere in the interval

$$T_A^r \in [T_B - \pi + d_{BA}^-(1 - \rho_A^-), T_B + P_{puzzle} + \pi + d_{BA}^+(1 + \rho_a^+)),$$

where π is the precision, d_{BA}^+ resp. d_{BA}^- is the upper resp. lower bound on the transmission delay from Bob to Alice, and ρ_A^+ resp. ρ_A^- is the upper resp. lower bound on the drift of Alice's clock. However, her acceptance interval for the next puzzle at time $T_B + P_{puzzle}$ will overlap with this one, creating a window

$$[T_B + P_{puzzle} - \pi + d_{BA}^-(1 - \rho_A^-), T_B + P_{puzzle} + \pi + d_{BA}^+(1 + \rho_a^+))$$

during which both puzzles are valid. Here, Alice should simply solve the first puzzle she receives, even if she obtains a more recent puzzle while working on the outdated one. As a consequence, Bob as the generator of the puzzle must accept solutions to a puzzle with timestamp T_B during the interval

$$[T_B + 2d_{min}(1 - \rho_B^-), T_B + P_{puzzle} + \pi + (2d_{max} + T_k)(1 + \rho_B^+)),$$

where $d_{min} = \min\{\min_A\{d_{BA}^-\}, \min_A\{d_{AB}^-\}\}$, $d_{max} = \max\{\max_A\{d_{BA}^+\}, \max_A\{d_{AB}^+\}\}$, and T_k is the maximum computation time for a puzzle of difficulty level k. There is again a time of overlap during which Bob must accept solutions to an old puzzle even though he has already generated a new one. But this is no DoS attack opportunity since Bob only stores data for correct puzzle solutions. Since Bob allows this overlap and includes Alice's clock synchronization uncertainty π into it, Alice is safe from DoS attacks via old puzzles since the solution to every puzzle she accepts as valid will also be accepted by Bob as long as Alice does not take more time than T_k to solve the puzzle.

If clock synchronization is not available, however, then Alice must accept every puzzle sent to her and will always have to solve the puzzle with the highest sequence number. Therefore, she must discard a puzzle she is working on if she receives a puzzle with a higher sequence number. Bob's window of acceptance will remain the same, but with π set to zero if Bob's timestamp is invalid.

Since we rely on timestamps and sequence numbers, we must consider what happens after a wrap-around. In the case of timestamps, it is clearly the task of clock synchronization to anticipate the wrap-around and mark timestamps as invalid until the precision condition is fulfilled again. Since we are prepared for transient losses of clock synchronization anyway, the wrap-around of the timestamp itself is of no particular concern. There is, however, the problem that for every point in time in the new timeline, there are puzzles which were acceptable in previous timelines. If Alice's timestamp and the timestamp of the old puzzle are both valid, Alice will accept the old puzzle and Bob will discard the solution. For this attack to work, the intruder must have the old puzzles (in fact, as large a sequence of them as possible to have more than a minor impact), timestamps must be valid in the old puzzles and now, the old timestamps must match Alice's acceptance windows, the signature must still be valid, and the intruder must know the current nonce Alice has sent to Bob. The attack lasts as long as the intruder has old puzzles with suitable timestamps and signatures. Although possible, we deem the chances for this attack to work to be rather low. The particular time at which a wrap-around occurs depends on the protocol used, NTP, e.g., has a 136-year cycle and the next wrap-around will occur in the year 2036 [41].

Sequence numbers are a fallback mechanism in case timestamps are invalid. Here, a wrap-around has severe consequences in that an intruder can replay any previous puzzle containing a higher sequence number than the current one and thus can cause Alice to reject the correct puzzle. As before, this only works if the signature of the old puzzle is still valid, i.e., if Bob has not changed his key since then. For the attack to succeed, the intruder only has to store one old puzzle, preferably with an invalid timestamp, that has a high sequence number, and can then prevent Alice from solving the correct puzzle until clock synchronization is available. But fortunately, the problem is not too pressing either: If we assume that sequence numbers are stored in a 16 bit variable and if Bob generates a new puzzle every hour as suggested in [4], then we obtain an 8-year cycle before the sequence number wraps. Bob should change his key more frequently anyway. However, Bob should store his puzzle sequence number in non-volatile memory to make sure he can recover it after a node crash.

4.2 Authenticated Communication

Let us assume that Alice and Bob want to talk and that they already know the fresh public key of the peer. Then we will use the algorithm sketched in Section 4.1 to set up authenticated communication with message authentication codes.

Algorithm 4 (Authenticated Communication)

1. $\mathrm{A} \rightarrow \mathrm{B}$: $[\mathrm{B}, \mathrm{A}, \mathrm{T}^s_A, \mathrm{T}^r_B]$, N_A, `puzzle request`
2. $\mathrm{B} \rightarrow \mathrm{A}$: $[\mathrm{A}, \mathrm{B}, \mathrm{T}^s_B, \mathrm{T}^r_A]$, N_A, $\sigma_B(T_B, k, N_B, S_B)$
3. $\mathrm{A} \rightarrow \mathrm{B}$: $\sigma_A([\mathrm{B}, \mathrm{A}, \mathrm{T}'^s_A, \mathrm{T}'^r_B], \mathrm{N}_B, \mathrm{N}'_A, X, \{\mathrm{K}_{AB}\}_{K_B})$
4. $\mathrm{B} \rightarrow \mathrm{A}$: $\sigma^h_B([\mathrm{A}, \mathrm{B}, \mathrm{T}'^s_B, \mathrm{T}'^r_A], \mathrm{N}'_A, \mathrm{Data}_B)$

In the first two messages, Alice requests a puzzle and Bob sends it to her. In the third message, Alice sends Bob a shared secret key for MAC computations. Bob stores the key, thus telling his hardware that all further messages to Alice should include a MAC. After sending the third message, Alice also stores this key, enabling her hardware to verify all further MAC messages sent by Bob. With the fourth message, Bob already includes a MAC and thereby acknowledges that he has received the MAC key. Bob may also include payload data if the communication is two-way. In all further message exchanges, MACs are used for authentication.

For one-way communication from Alice to Bob, Alice will have to store the following parameters for Bob:

- Bob's name
- Bob's public key (only if data should be encrypted)
- shared MAC key (if MACs are used)

After Alice has been authenticated and the communication request has been granted, Bob as the receiver will also have to store some parameters:

- Alice's name
- message filter mode
- Alice's public key (if signatures are used)
- shared MAC key (if MACs are used)
- delay parameters per path from Alice to Bob (if timestamp checking is used)

In a two-way communication, Alice stores the same items as Bob.

All DoS attacks on the puzzle request have already been described in Section 4.1. We see that in a one-way communication, Bob as the receiver has to store more data than Alice as the sender. Alice can even renounce timestamp filtering and just use a software filter on the nonce to discard replayed messages from previous protocol runs while waiting for Bob's last reply. She also does not have to store a message filter mode, since the hardware automatically uses MAC filtering if the message type indicates a MAC and if an appropriate key is stored. Nevertheless, an intruder does not gain anything from establishing an unnecessary connection with Bob since the intruder will have to solve the puzzle first and will be challenged with a puzzle again if Bob starts to become overloaded.

After MAC authentication is established, only messages bearing acceptable or invalid timestamps will pass through the message filters, all other messages are discarded by the hardware during reception. Nevertheless, due to the uncertainties involved in clock synchronization and the transmission delay estimation, there is still an opportunity for replays here, so the software responsible for the payload data should use sequence numbers if necessary.

Note that whereas we can safely assume that a node always has enough CA servers in its vicinity to rely on a synchronous subsystem, we cannot always be sure that Alice and Bob are within such a subsystem. We argue that it is the general case, but we must also consider what happens if Alice and Bob are so far apart that we must assume an asynchronous system. In this case, our assumptions on network connectivity and transmission delay bounds may not hold. The security of the authentication protocol is nevertheless guaranteed: Since both Alice and Bob have a working connection to the CA, they can get the fresh public key of the peer to verify the signatures. And since only Bob can decrypt the MAC key Alice has sent to him, all further communication using MACs is also authenticated. There is, however, more opportunity for DoS attacks since communication may be disrupted for an arbitrary amount of time. Therefore, the windows for DoS attacks on Alice during the puzzle exchange phase will become much larger. What is more, Alice's solution to a puzzle may be delayed long enough to arrive only after Bob has already discarded the puzzle. Although Bob should choose his puzzle generation period P_{puzzle} large enough to minimize the chances of such a thing happening, nevertheless communication may be interrupted for an arbitrarily long time.

4.3 Key Distribution

If we use a public key system, then we must assume that a trusted certification authority (CA) knows the public keys and can verify the binding of public keys to nodes. We assume a fault-tolerant distributed CA like COCA [80, 78]. Note that our system assumptions concerning links are stronger than those of COCA in that we assume at least one correct path, whereas COCA only requires links to be fair, i.e., if infinitely many messages are transmitted, then infinitely many messages are correctly delivered. Apart from that, they assume a fully asynchronous system, whereas we require a synchronous subsystem to implement our denial-of-service defenses. The assumptions complement one another nicely, though, since the CA itself is distributed over the whole network and hence cannot depend on synchronicity, whereas a node communicating with the CA does so by talking to nearby servers that are part of a local subsystem in which the synchronicity assumption is viable. In both our system and COCA, if clients are partitioned for too long, then either keys must be considered as out-dated and system operation is disrupted, or clients must take the risk of using compromised keys.

In key publication, our concern is how to deliver fresh public keys to the nodes who need them. In order to avoid constant polling of the certification authority whenever a message arrives, we chose a hybrid approach. If Bob needs to check a single message from Alice and does not have a fresh key, he requests Alice's key certificate from the CA. The request does not include signatures, we just assume that the CA can handle many requests and will start sending puzzles if overloaded. Instead of a single request, Bob can also send a subscription request, which makes the CA send Bob the new certificate whenever Alice's key changes. Such subscription requests must be signed.

The protocol for a single query without authentication is very simple and consists of just two messages:

Algorithm 5 (Single Query Key Distribution)

1. B $\rightarrow$ CA: [CA, B, T^s_B, T^r_{CA}], A, N_B, `request certificate`
2. CA $\rightarrow$ B: [B, CA, T^s_{CA}, T^r_B], $\text{Cert}_A(K_A) = \sigma_{CA}(A, K_A, \nu_A)$

In this simple query protocol, no signature is required from Bob under normal circumstances, since the CA has a lot of capacity anyway and can afford to answer all requests. The reply message itself is not signed to allow the CA to precompute the certificate. If, however, the CA gets overloaded or if it suspects that a DoS attack has been launched, then it will reply with a puzzle instead of the certificate. For this reason, Bob includes a nonce N_B with the request and stores it until he receives a message from the CA.

If Bob wants to start or cancel a subscription to certificate updates, or if he wants to be sure that the certificate he receives is fresh and from the CA, then he must authenticate himself and therefore must request a puzzle first, resulting in the following protocol:

Algorithm 6 (Authenticated Key Distribution)

1. B $\rightarrow$ CA: [CA, B, T^s_B, T^r_{CA}], N_B, `puzzle request`
2. CA $\rightarrow$ B: [B, CA, T^s_{CA}, T^r_B], N_B, $\sigma_{CA}(T_{CA}, k, N_{CA}, S_{CA})$
3. B $\rightarrow$ CA: σ_B([CA, B, T'^s_B, T'^r_{CA}], N_{CA}, N'_B, X, A, $\{K_{B,CA}\}_{K_{CA}}$, `once|subscribe|stop`)
4. CA $\rightarrow$ B: σ^h_{CA}([B, CA, T'^s_{CA}, T'^r_B], $\text{Cert}_A(K_A)$)

4′. CA $\rightarrow$ B: σ^h_{CA}([B, CA, T'^s_{CA}, T'^r_B], `subscription denied|stopped`)

In the first message, Bob sends his request to the CA, and the CA replies with the puzzle in the second message. Bob sends the solution in the third message, along with his request. Bob can request a single certificate, subscribe, or cancel an existing subscription. In any case, he includes a secret MAC key for further communication with the CA. If Bob has requested a certificate or a subscription, the CA will reply with the certificate in message (4). This message includes the MAC and therefore ensures Bob that the message is fresh and from the CA. Once Bob has subscribed, the new certificate is automatically sent to him whenever Alice's key changes. These certificate update messages look like message (4).

If Bob wants to cancel a subscription, he will receive an acknowledgment with message (4′). This message is also used in cases where for some reason the CA cannot process a subscription request (most likely due to overload).

The subscription feature is used to place the load of keeping Bob up-to-date with Alice's current key on the CA, which has most likely better hardware than Bob. In consequence, the nodes do not have to query the CA as often, so if key changes are not very frequent, both the CA and the nodes are relieved from message traffic. In turn, it requires some memory for storing the subscription on the CA. If memory is of concern, the CA can force a node to resubscribe regularly by only storing subscriptions for a limited period of time.

The CA responds to Bob's request with Alice's current key certificate. Only the very first key certificate message is passed on to Bob's software, which determines whether the certificate should be processed and in that case orders the hardware to store the data from the certificate. As soon as a key is stored for a node, all subsequent certificates are processed within the W2F box. Upon reception, Bob's W2F box checks whether the version number contained in the certificate is higher than the one currently stored, and in that case verifies the certificate and extracts the new key from it. Processing is done automatically by the hardware to relieve the software from a task that ultimately results in the hardware processing the certificate anyway. The software protocol only has to decide whether the key for a particular node should be stored in the first place, and it also determines when the key of a node should be removed from the key table.

Since the key of the CA is extremely important and must not be compromised at all costs, the CA itself should change its key from time to time. It will broadcast its new public key (using the old key to sign the message) to all nodes in the network. As long as every node is connected over a correct path to the CA, every node will get the new key and will update its key table. After the broadcast, the CA can use the new key.

Apart from DoS attacks on Bob during the puzzle request phase, which we have described in Section 4.1, we do not see any other attack opportunities on Bob in the authenticated protocol since he is protected by the MAC. We also do not see any attacks against the CA except that Bob can make the CA store his subscription (two names and a MAC key) and send messages to him whenever the key he has subscribed to changes. To minimize abuse, subscriptions should expire after some time.

The unsigned certificate message sent by the CA in the unauthenticated version allow an attack on Bob by sending faked certificates with a high version number. Bob has no defense against that and must check the signature of each such message. If Bob fears he is under attack, he should use the authenticated protocol and ignore all unsigned certificate messages sent to him.

An attack on the authentication itself can be launched if the certificate version number wraps before the CA key changes, because now the intruder can send any old certificate with a higher number. But as we have argued for the sequence numbers described in Section 4.1, the cycle for a wrap-around is much longer than the key change rate anyway.

4.4 Key Revocation

Frequent key changes help to prevent an attacker from guessing a fresh key with a brute-force attack. Therefore, all nodes should periodically change their keys. Here, we are faced with the dilemma of who to entrust with generating the new keys. If we place the responsibility on the CA, then we can be sure that it will select good, i.e., cryptographically secure, keys. However, if just one server of the CA is corrupted, it may find out the secret keys of all nodes. On the other hand, if we let the nodes generate their own keys, then not every node may have enough computing power to select a good key. Still, it is a trade-off between some small nodes using insecure keys and having to ensure that no server node of the CA is ever corrupted to secure the keys of all nodes. We strongly suggest to let nodes choose the keys for themselves, but nevertheless we provide algorithms for both variants.

In the *server-based key revocation* protocol, new keys are issued by the CA on the basis that the CA has enough computing power to select good keys, so Alice must request a key update from the CA to change her key. Even if Alice is still uncompromised, the CA cannot use her current public key to encrypt the new private key: An attacker who finally cracks one old key and who possesses all subsequent key change messages sent by the CA can then extract the current key as well. Therefore, we must resort to a different key shared between the CA and Alice. We will use the Diffie-Hellman protocol [14] for generating the shared key, which relies on the difficulty of computing from a given X, g, and n the x for which holds $X \equiv\ g^x\ mod\ n$. The protocol assumes that two parties select a large prime n and some value g that is primitive mod n. These two values are public knowledge. To establish a common shared key between Alice and Bob, Alice chooses a large random number x and sends the value $X = g^x \bmod n$ to Bob. Bob, in turn, selects some large random number y and sends Alice the value $Y = g^y \bmod n$. Since $k = Y^x = X^y = g^{xy} \bmod n$, Alice and Bob now share a key k that cannot be computed by anyone else.

To integrate this protocol into our system, we assume that Alice selects some large random number x prior to sending the key update request and sends $g^x \bmod n$ to the CA together with the puzzle solution. Upon verification of the solution, the CA generates a new public/private key pair for Alice, selects some value y, computes the shared key $K_{A,CA} = g^{xy} \bmod n$ to encrypt Alice's new key pair, and sends its public value $g^y \bmod n$ together with Alice's encrypted new keys. Upon reception, Alice uses g^y to compute the shared key and decrypts her new key pair, which is loaded into the W2F box. The protocol for requesting the key update looks like this:

Algorithm 7 (Server-Based Key Revocation)

1. A → CA: $[\mathrm{CA}, \mathrm{A}, \mathrm{T}^s_A, \mathrm{T}^r_{CA}]$, N_A, `puzzle request`
2. CA → A: $[\mathrm{A}, \mathrm{CA}, \mathrm{T}^s_{CA}, \mathrm{T}^r_A]$, N_A, $\sigma_{CA}(T_{CA}, k, N_{CA}, S_{CA})$
3. A → CA: $\sigma_A([\mathrm{CA}, \mathrm{A}, \mathrm{T}'^s_A, \mathrm{T}'^r_{CA}], \mathrm{N}_{CA}, \mathrm{N}'_A, X, g^x \bmod n,$ `key update`$)$
4. CA → A: $\sigma_{CA}([\mathrm{A}, \mathrm{CA}, \mathrm{T}'^s_{CA}, \mathrm{T}'^r_A], \mathrm{N}'_A, g^y \bmod n, \{\ \mathrm{K}'_A, \mathrm{K}'^{-1}_A\ \}_{K_{A,CA}})$

Note that for the CA to authenticate Alice with the third message, we must assume that Alice's key is still uncorrupted.

As soon as Alice sends her public part of the shared secret key, an attacker can work on finding her secret value x. If the attacker finds that value, it can compute the shared key and read Alice's secret key. Therefore, we must require that Alice requests a new update earlier than that. Note that decrypting an old key update reply does not help the attacker to find recent key values, since Alice and the CA use a new random key with every update request.

In the *client-based key revocation* protocol, Alice is allowed to change the key by herself, so she just wants the CA to sign and store her new public key certificate and does not require a shared key.

Algorithm 8 (Client-Based Key Revocation)

1. A → CA: [CA, A, T^s_A, T^r_{CA}], N_A, `puzzle request`
2. CA → A: [A, CA, T^s_{CA}, T^r_A], N_A, $\sigma_{CA}(T_{CA}, k, N_{CA}, S_{CA})$
3. A → CA: σ_A([CA, A, T'^s_A, T'^r_{CA}], N_{CA}, N'_A, X, K'_A, `key update`)
4. CA → A: σ_{CA}([A, CA, T'^s_{CA}, T'^r_A], N'_A, $\mathrm{Cert}_A(K'_A)$)

Both protocol variants have in common that if Alice's signature key has already been compromised, then anyone can request a key update in Alice's name if they commit their resources to the puzzle. Since the new key pair or the certificate will be sent to Alice, she will at least know that her key has been compromised and can raise an alarm. In the worst case, however, Alice's W_2F box will have to be re-initialized and her new public key will have to be entered into the CA manually.

4.5 Assumption Violations

As we have already mentioned, W_2F is based on an ad-hoc network. Therefore, the assumptions we stated at the beginning of this chapter may not always hold, and we must consider what happens if they are violated. Basically, we have three assumptions to consider:

(A1) The key of the certification authority is not compromised.

(A2) Nodes are connected by at least one correct path.

(A3) The system is synchronous.

Assumption (A1) is by far the most important of all since its violation affects the whole system. If an intruder breaks the key of the certification authority, then it can generate new certificates containing fake keys for any node at will, corrupting authentication at least until the next key update from the real CA comes through. At that point, corruption of the CA will at least be detected. Unfortunately, as long as we cannot identify and remove the intruders and all knowledge of the key, we fear that only manual change of the CA key and corresponding reconfiguration of all nodes will serve to re-establish authentication.

Violation of (A2) may lead to network partitions. Any nodes that are separated from the CA will not get new key updates and may work with old keys. To limit this period, nodes should use a timeout after which a certificate becomes invalid. This is also true for the key of the CA if (A1) can happen. In this case, reconfiguration after the rejoin must include high-level consensus mechanisms to establish the correct public key of the CA. A majority vote must be computed over the stored public CA keys of a sufficient number of unpartitioned nodes. If we assume that n_p nodes were partitioned from the CA and if there are at most f intruders and faulty nodes, then we need $n > 2f + n_p$ nodes in the system to obtain the correct majority.

If (A3) is violated, then the timestamp filter is useless. As long as the violation is recognized, we can simply turn it off, but if the problem goes undetected, then valid messages are filtered out, creating an effect similar to violation of (A2) with all its consequences. In any case, this will significantly increase the DoS attack opportunities during client puzzle exchanges.

Here, we must decide between quality of service and security: If we want high quality of service and if undetected violation of (A3) is highly unlikely, then we should use the timestamp filter. If our ultimate goal is as secure authentication as possible and if the probability of undetected violation of (A3) is unacceptably high, then we must not use the timestamp filter and will experience more impact from DoS attacks. If the synchronicity condition is only violated sometimes, then timeouts and a simple retransmission scheme will suffice to ensure normal operation.

Chapter 5

Consensus

Reaching *consensus* among the processes of a distributed system is widely recognized as one of the most fundamental problems in fault-tolerant distributed computing [34]. Unfortunately, there is a discouraging impossibility result for deterministic consensus in presence of link failures, which goes back to Gray's 1978 paper [27] on atomic commitment in distributed databases (see Section 5.2.2 for its proof):

Theorem 1 (Gray's Impossibility [37, Thm. 5.1]) *There is no deterministic algorithm that solves the coordinated attack problem in a synchronous two-process system with lossy links.*

Due to this result, almost all deterministic algorithms for consensus developed during the past 20+ years considered process failures only. Our analysis revealed, however, that if the power of link failures is slightly restricted with respect to the inconsistencies they may cause system-wide, (most) existing consensus algorithms can be made resilient to a large number of link failures if the number of processes n is moderately increased.

We thus prove the correctness of the family of *Hybrid Oral* and *Written Messages* algorithms[1] for *Byzantine agreement* under our failure model and establish detailed formulas for the required number of processes.

All these algorithms are "Byzantine generals" algorithms where the value v of a dedicated *transmitter* is to be disseminated to the remaining $n-1$ *receivers*; the transmitter already knows its value. Eventually, every non-faulty process p—including the transmitter—must *deliver* a value v_p ascribed to the transmitter that satisfies the following properties:

(B0) (*Termination*): The algorithm terminates eventually.

(B1) (*Agreement*): If processes p and q are both non-faulty, then both deliver the same $v_p = v_q$.

[1]Note that we are aware of the fact that those algorithms suffer from an exponential number of messages. Given that they are hybrid instances of the most well-researched algorithm [34] for Byzantine agreement, however, they are certainly the most suitable candidates for introducing our fairly general approach. More efficient (polynomial) consensus algorithms are treated in [7].

(B2) (*Validity*): If process p is non-faulty, the value v_p delivered by p is

- v, if the transmitter is non-faulty,
- E, if the transmitter is manifest faulty,
- v or E, if the transmitter is omission faulty,
- the value actually sent, if the transmitter is symmetric faulty,
- unspecified, if the transmitter is arbitrary faulty.

Consensus is very similar, but in contrast to Byzantine agreement, where just one transmitter disseminates its value, consensus is the problem of computing a common output value from a set of input values distributed among all processes. More specifically, every process p provides an *input value* x_p, which is supplied to the local instance of a distributed consensus algorithm simultaneously at all nodes. Within a finite number of rounds (that may be different for different processes, and may even depend upon the particular execution), p irreversibly computes ("decides upon") an *output value* y_p, which must satisfy the following properties:

(C0) (*Termination*): The algorithm terminates eventually.

(C1) (*Agreement*): If processes p and q are both non-faulty, then both compute the same output value $y_p = y_q$.

(C2) (*Validity*): If all processes p start with the same input value $x_p = x$, then every non-faulty process q computes the output value $y_q = x$.

Sometimes it is useful to consider *binary consensus*, where both input and output values are binary, i.e., $x_p, y_p \in \{0, 1\}$.

A fully-fledged consensus algorithm can be obtained by using a separate instance of Byzantine agreement for disseminating any process's local value and using a suitable choice function (majority) for the consensus result.

5.1 System Model

We consider a distributed system of n *nodes*, which execute one or more *processes* that communicate via a fully connected point-to-point network. In case of multiple processes per node, it is assumed that every such process has a unique peer at every other node. All processes executed by a non-faulty node must be non-faulty, zero or more processes executed by a faulty node may be faulty as specified.[2] All links between processes are bidirectional, consisting of two unidirectional channels that may be hit by failures independently.

The entire system is synchronous in the usual sense, that is, the distributed computation evolves in a series of rounds at all processes in lock-step. In every round, all processes (or a subset of those) "broadcast" a single value to each other; the received

[2] The terms process and node failures can hence be used more or less synonymously. We prefer the more common term process failure.

values are used to compute the value to be broadcast in the next round. Broadcasting just means non-atomic, non-reliable sending of the same message to all receivers (including the transmitter) here. In case of failures, inconsistent reception of a broadcast message may hence occur. It is the purpose of the failure model to cleanly specify the failures that are to be tolerated by the distributed algorithm in question.

The *perception-based hybrid failure model* introduced below is a generalization of the failure model used for the analysis of clock synchronization and other single-round agreement algorithms in [55]. In this model, the global, i.e., system-wide, number of failures is replaced by the number of failures that are observable in the processes' local "perceptions" of the system. Formally, process r's *perception vector*

$$\boldsymbol{\mathcal{V}}_r = (V_r^1, V_r^2, \ldots, V_r^n) \tag{5.1}$$

of any specific round R is considered (we suppress the round number R for clarity), where every *perception* $V_r^s \in \boldsymbol{\mathcal{V}}_r$ represents the message process r received from its peer process s in round R; type and value(s) depend upon the particular algorithm considered. For approximate agreement algorithms, for example, the V_r^s would be real values that represent the receiver's opinion about the sender's local value V^s.

In case of the single-round algorithm analyzed in [55], we found it sufficient to just impose a bound upon the maximum number of failures in any *pair* of perception vectors $\{\boldsymbol{\mathcal{V}}_p, \boldsymbol{\mathcal{V}}_q\}$, i.e., p's and q's lines in the "matrix" of perceptions on the right-hand side of

$$\begin{aligned}
\boldsymbol{\mathcal{V}}_1 &= (V_1^1, V_1^2, \ldots, V_1^n) \\
\boldsymbol{\mathcal{V}}_2 &= (V_2^1, V_2^2, \ldots, V_2^n) \\
&\vdots \\
\boldsymbol{\mathcal{V}}_n &= (V_n^1, V_n^2, \ldots, V_n^n).
\end{aligned}$$

For example, if every process may lose at most $f_\ell = 1$ messages due to faulty links (see Figure 5.1), any two non-faulty processes' perception vectors can differ only in at most $2f_\ell = 2$ perceptions, namely, the ones where either receiver process experienced its omission. Moreover, only at most $f_\ell = 1$ of the non-faulty perceptions present at some non-faulty process can be missing or faulty at any other non-faulty process. Even more, since f process failures can produce at most f faulty perceptions in any $\boldsymbol{\mathcal{V}}_r$, our perception-based model is compatible with traditional process failure models. Hence, all existing lower bound and impossibility results remain valid.

Depending upon the type of failure of a perception, e.g., missing or value faulty, several different classes of failures (manifest/omission/symmetric/asymmetric) can be distinguished. This leads to a *hybrid failure model*, see [5] for an overview, which allows to exploit the fact that less severe failures can usually be handled with fewer processes than more severe ones. In case of consensus, for example, masking f symmetric failures usually requires only $n \geq 2f + 1$ processes, whereas $n \geq 3f + 1$ is needed if all failures are asymmetric (Byzantine) ones. Since a large number of asymmetric failures is quite unlikely in practice, this effectively leads to a smaller n for tolerating a given number of failures. This, in turn, has a positive effect upon dependability by reducing the number

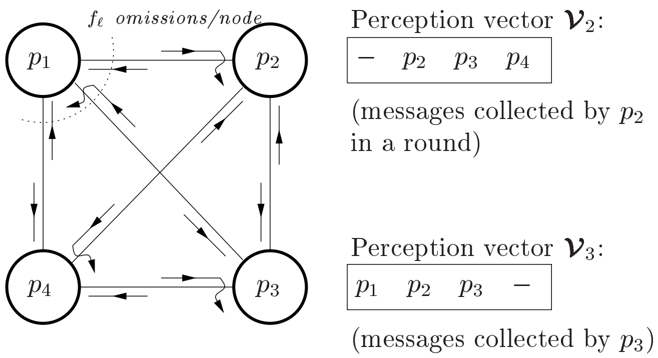

Figure 5.1. *Example of a 4-process system with $f_\ell = 1$ receive failures per process in each round.*

n of components that could be faulty, cf. [47]. System designers will hence appreciate our very detailed hybrid failure model for getting the maximum fault-tolerance out of a given—and usually quite small—n. Obviously, an algorithm's resilience in standard models (like all-Byzantine) is easily[3] obtained by setting some model parameters to 0.

For Srikanth & Toueg's consistent broadcasting primitive [67] in asynchronous systems, for example, the analysis in [55] revealed that

$$n \geq 4f_{ia} + 3f_a + 2(f_s + f_{is} + f_o + f_{io}) + f_m + 1$$

processes are sufficient for tolerating at most f_m, f_o, f_s, f_a crash, omission, symmetric, and arbitrary process failures and at most f_{io}, f_{is}, and f_{ia} omission, symmetric, and arbitrary link failures at each receiver process.

In view of the results of [55], it was natural to consider the question of whether a perception-based failure model can also be used to attack deterministic consensus in presence of link failures. More specifically, as the general problem is unsolvable by Theorem 1, one might ask whether there is a meaningful restriction of the power of link failures that can be expressed in a perception-based manner. And indeed, as will be shown in the subsequent sections, there is a suitable perception-based failure model that allows even consensus with oral messages [34] in systems with high link failure rates. Ignoring process failures for the moment, it is based upon constraining two quantities in the matrix of perceptions of a single round:

(A1^r) There may be at most f_ℓ^r columns with erroneous perceptions in any fixed single line. This puts a limit upon the number of sender processes that may appear faulty to a single receiver (already employed in [55]), see Figure 5.2. (A1^r) thus restricts the number of *receive link failures* per process to f_ℓ^r. A process that suffers from at least one receive link failure is said to commit a *receive failure*.

[3]Note, however, that the general hybrid analysis might be too conservative for certain restricted cases, see Remark 4 on Theorem 8 for an example.

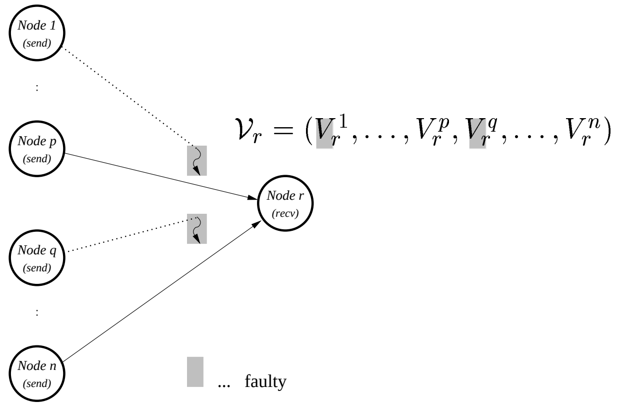

Figure 5.2. *Example of a process that suffers from two receive link failures.*

($A1^s$) There may be at most f_ℓ^s lines with erroneous perceptions in any fixed single column. This puts a limit upon the number of receiving processes that may obtain a wrong (or no) message in the broadcast of a single sender, see Figure 5.3. ($A1^s$) thus restricts the number of *send link failures* (also called *broadcast link failures*) per process to f_ℓ^s. A process that suffers from at least one broadcast link failure is said to commit a *broadcast failure*.

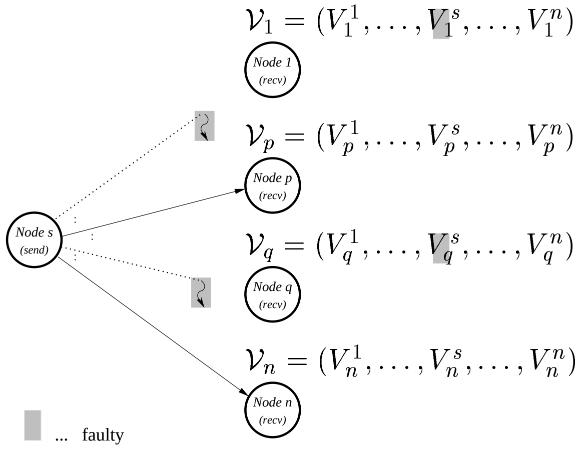

Figure 5.3. *Example of a process that suffers from two send link failures.*

Note carefully that we allow <u>any</u> process in the system to suffer from up to f_ℓ^s send resp. up to f_ℓ^r receive link failures in any broadcast resp. reception, without considering the process as faulty in the usual sense. Moreover, the particular links actually hit by broadcast resp. receive link failures in any two different broadcasts resp. receptions in the same and in different rounds are usually different.

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

Nevertheless, our failure model considers ($A1^s$) and ($A1^r$) as independent of each other and of process failures, for any process in the system and any broadcast/reception. Only the model parameters f_ℓ^s and f_ℓ^r do depend on each other: According to ($A1^s$), at most $n \cdot f_\ell^s$ messages may be faulty system-wide in any round's broadcasts. By ($A1^r$), they must be spread over all message receptions in a way that no process experiences more than f_ℓ^r faulty messages. This is only possible if $r = n \cdot f_\ell^r = n \cdot f_\ell^s = s$, however: If $s > r$ and hence $f_\ell^s > f_\ell^r$, then there would be at least one receiver that gets more than f_ℓ^r faulty messages and thus violates ($A1^r$); if $s < r$ and hence $f_\ell^s < f_\ell^r$, then there would be at least one sender process with more than f_ℓ^s broadcast link failures, thereby violating ($A1^s$).

Consequently, since both ($A1^s$) and ($A1^r$) must[4] hold in any execution, there are only two possibilities: (1) $f_\ell^s = f_\ell^r = f_\ell$, in which case ($A1^r$) and ($A1^s$) always hold, or (2) $f_\ell^s \neq f_\ell^r$, in which case the failure model only applies if link failure patterns are restricted to feasible ones.

To align the failure model with the existing literature, the above perception-based model is recast into a simple modification of the original oral messages assumption of [34]. Process failures are modeled according to a generalized version of the hybrid failure model of [35]. Our contribution here is to add the important class of *process omission failures* (= *send/receive omissions* [45]) to the originally provided manifest, symmetric, and arbitrary failures. In doing so, those frequently encountered process failures [5] need not be counted as arbitrary, which further decreases the required number of processes n (but see Remark 4 on Theorem 8).

In order to cleanly specify the semantics of restricted failures, we will need the notion of *obedient* processes: An obedient process is an operational (= not crashed) process that gets its inputs and faithfully executes the particular algorithm like a non-faulty process. Unlike a non-faulty process, however, it may fail in specific ways to communicate its value to the outside world. For example, an obedient but not non-faulty process could omit to send its (correctly computed) value to some subset of its receivers. Note that Definition 2 will actually restrict the meaning of obedient to non-faulty or manifest faulty or omission faulty (but not crashed).

We will subsequently use the term obedient instead of non-faulty whenever a process is considered from a receiver's perspective only. Note carefully, however, that a generalization from non-faulty to obedient processes may not always work, cp. algorithms OMH vs. OMHU in Section 5.3 for an example.

Definition 2 (System Model) *We consider a synchronous distributed system consisting of n nodes executing one or more processes, which are interconnected by a fully connected point-to-point network made up of pairs of unidirectional channels.*

[4] Note carefully, however, that there are algorithms that do not need ($A1^s$) and ($A1^r$) simultaneously in any single round, but only one of those, see Remark 5 on Theorem 8.

(P1) *In any entire execution, there may be at most f_a, f_s, f_o, and f_m arbitrary, symmetric, omission, and manifest faulty nodes. The failure modes of their processes are defined via the set rvals(V_p, p) of admissible values delivered by obedient receivers when process p attempts to send them the value V_p:*

- *A* manifest *faulty process p fails to send a message, or sends an obviously bad value, to any receiver. All obedient receivers q in the system (including p itself) deliver the distinguished value $V_q^p = \emptyset$ in this case, i.e., rvals$(V_p, p) = \{\emptyset\}$.*
- *An* omission *faulty process p may fail to send the correct value V_p to some of its obedient receivers q_i, which deliver $V_{q_i}^p = \emptyset$ instead of V_p in this case. Hence, rvals$(V_p, p) = \{V_p, \emptyset\}$.*
- *A* symmetric *faulty process p sends the same wrong (but not usually obviously bad) value X_p to every receiver q. All obedient receivers (including p itself) deliver $V_q^p = X_p$—the value* "actually sent"*—in this case, such that rvals$(V_p, p) = \{X_p\}$.*
- *An* arbitrary (asymmetric) *faulty process may inconsistently send any value to any receiver, so rvals(V_p, p) is the set of all possible values, including $\emptyset$.*

A process that is manifest or omission faulty is called benign *faulty and is assumed to be obedient (if not crashed). A process is* consistent *if it is either non-faulty, manifest faulty, or symmetric faulty.*

(A1^s) *If a single [faulty or non-faulty] process p sends a value V_p to some set of obedient receiver processes $q_i \in \mathcal{R}$, at most f_ℓ^s of the delivered values $V_{q_i}^p$ may differ from the admissible receive values in rvals(V_p, p). Let $f_\ell^{sa} \le f_\ell^s$ be the maximum number of non-omissive, i.e., non-empty and hence value faulty, $V_{q_i}^p$ among those.*

(A1^r) *If all processes $p_i \in \mathcal{S}$ of a set of [faulty or non-faulty] processes send a message containing V_{p_i} to some obedient receiver process q, at most f_ℓ^r of the delivered values $V_q^{p_i}$ may differ from the admissible receive values in rvals(V_{p_i}, p_i). Let $f_\ell^{ra} \le f_\ell^r$ be the maximum number of non-omissive, i.e., non-empty and hence value faulty, $V_q^{p_i}$ among those.*

(A2) *The receiver of a message knows who sent it.*

(A3) *The absence of a message from sender p can be detected at any receiver q at the end of a round, which leads to $V_q^p = \emptyset$.*

Remarks:

1. A single round consists of (1) all the processes' local computations based upon the values received in the previous round, (2) the broadcasts (= successive sends) of the resulting messages according to (A1^s), and (3) the reception of those messages according to (A1^r).

2. Properties (A1^s) and (A1^r) must hold simultaneously in any feasible execution and are assumed to be independent of each other. Any receive resp. broadcast

failure has an individual "budget" f_ℓ^r resp. f_ℓ^s of link failures that may hit arbitrary inbound resp. outbound links. Without restricting link failure patterns, however, this can only be guaranteed if $f_\ell^s = f_\ell^r$ and $f_\ell^{sa} = f_\ell^{ra}$.

3. The system model of Definition 2 considers process and link failures independently. Therefore, even a manifest or omission faulty process's broadcast could generate erroneous values at f_ℓ^{sa} receivers, for example. By contrast, the original model in [57, Def. 1] assumed that link failures hit only messages from non-faulty senders/receivers. The new model is more natural and has a better coverage in real systems, but requires a slightly more involved analysis.

4. Faulty processes must not change their failure mode, i.e., must be counted in f_a, f_s, f_o or f_m according to their most severe behavior. Recall that those numbers specify the maximum number of faulty *nodes* in the entire execution. Hence, a node that hosts one or more processes that e.g. behave symmetric faulty in some round and omission or manifest faulty in some round must be considered arbitrary faulty.

5. Our manifest failures differ from the systemwide detectable ones of [47] and the "benign failures" of [5, 71] by also including symmetric omissions (produced e.g. by clean crashes), where no receiver gets a non-$\emptyset$ value. Since a receiver does not know locally whether a message is missing due to a symmetric omission or e.g. a Byzantine failure, symmetric omissions are usually more difficult to handle than "benign failures". This is not true for the algorithms analyzed here, however, so integrating both into manifest failures as in [35] makes sense.

6. Our failure model provides only *strong* manifest failures, where all receivers, including the sender itself, get $\emptyset$-values only. Some related work like [51] considers *weak* manifest failures, where the sender is allowed to deliver the correct value instead of $\emptyset$, thereby causing some asymmetry in the reception. Of course, weak manifest failures can always be incorporated as omission failures in our model. Alternatively, they are equivalent to strong manifest failures if at least one arbitrary link failure may also occur (i.e., if $f_\ell^{ra} = f_\ell^{sa} \geq 1$): Self-delivery of the correct (even incorrect) value sent by a manifest faulty process can then be explained by a spurious message generated on the link to itself.

7. Our failure model does not contain a direct equivalent for standard *crash failures*, where a process can die once and forever even during its broadcast. Crash failures are in fact more severe than manifest failures but weaker than omission failures [45]. Nevertheless, both manifest and omission failures are more severe than crashes in that faulty processes may resume correct operation in any later round. Bear in mind, however, that this behavior is not equivalent to the crash-recovery model of [2], since our processes may not lose state but must continuously follow the algorithm, see Remark 8 below.

8. Benign faulty processes are allowed to convey either the correct value or else $\emptyset$ to their receivers, which implies that they must know the correct value at least internally. Although we need not care how this is actually accomplished, it is nevertheless true that the only way to ensure this in practice is to assume that benign faulty processes are obedient (or have crashed).

9. Arbitrary faulty processes need not adhere to the particular algorithm. Unlike a symmetric or benign faulty process, an arbitrary faulty process could even send multiple messages in a single round; a receiver may deliver any of those or $\emptyset$ in this case.

Since the failure model of Definition 2 allows link failures to be both transient and permanent, it is possible to model incomplete communication graphs by considering missing links as faulty, cp. [64]. The following Theorem 2 will show, however, that sparsely connected graphs and, in particular, partitioned ones are disallowed here. Recall from elementary graph theory that a graph G is *c-connected* if it remains connected by removing at most c processes and their adjacent edges. Two paths connecting processes p and q are *process-disjoint* if and only if they do not have common processes except p and q.

Theorem 2 (Connectivity) *The communication graph of any system of $n > f_\ell^s + f_\ell^r$ processes complying to the system model of Definition 2 is c-connected with $c = n - f_\ell^s - f_\ell^r > 0$. Moreover, any pair of processes p, q is connected by c process-disjoint paths consisting of at most 2 non-faulty links.*

Proof: We use Menger's theorem, which says that G is c-connected if and only if any pair of processes p, q is connected by c process-disjoint paths. To show the latter, we argue as follows: From ($A1^s$) in Definition 2, we know that p is connected to at least $n - f_\ell^s$ processes (possibly including itself) via non-faulty links. From ($A1^r$), it follows that q is connected to at least $n - f_\ell^s - f_\ell^r = c$ of these via non-faulty links. Let $\mathcal{I}$ with $|\mathcal{I}| \geq c$ be the latter set of processes. If $p \notin \mathcal{I}$ and $q \notin \mathcal{I}$, then p and q are connected by c paths consisting of 2 non-faulty links routed over the processes in $\mathcal{I}$. Otherwise, there are only $c - 1$ paths of length 2 and a direct path from p to q, which are of course also process disjoint. $\square$

5.2 Theoretical Results

5.2.1 General Proofs

The proofs in the following subsections will often rely on the generic impossibility proof of Theorem 3. It utilizes the technique used in [20] for the famous FLP impossibility result which demonstrates that consensus is not possible in an asynchronous system if there is at least one crash failure. The proof is based on the notion of bivalent configurations, which basically identify situations in which an algorithm needs another round to decide. The FLP proof shows that for any algorithm, if a bivalent configuration exists in some round r, then there must be at least one successor configuration in round $r+1$ which is also bivalent. Hence, by induction there must be an infinite sequence of bivalent configurations where the algorithm will never be able to decide. We will first give a couple of definitions and then present our generic impossibility proof.

Consider some (deterministic) algorithm which proceeds in synchronous rounds. The algorithm broadcasts a message to all processes at the beginning of the round, then waits for incoming messages for some time, and finally processes the received messages, computing the message to send in the next round. Depending on the failure model chosen, a round may consist of any message exchange pattern out of a finite set $\mathcal{M}$ of possible ones. In a system with two processes p and q and at most one link failure per round, e.g., the possible message exchange patterns in a round are that both messages arrive, that the message $p \to q$ does not arrive, and that $q \to p$ does not arrive.

Definition 3 (Configuration) *We call the state of all processes at the beginning of a round $r \geq 1$ a* configuration *of the algorithm. The configuration at the beginning of round 1 is an* initial configuration*.*

Definition 4 (Execution Run) *A (finite or infinite) sequence σ of message exchange patterns is called a (finite or infinite)* execution run*. The length of the run symbolizes the number of rounds it takes to execute it. Applying σ with length $l < \infty$ to some configuration C at the start of round r will lead to a configuration C' at the start of round $r+l$. An execution run can also be empty, i.e., $l = 0$.*

Definition 5 (Successors and Neighbors) *Let C be some configuration. We call C' a* successor configuration *of C if it is obtained from C by executing one more round. Two successor configurations C' and C'' are* neighbors *if the single rounds leading to them from C only differ in one message which arrives in one configuration and is lost in the other. If their last rounds differ in k messages, then C' and C'' are neighbors of degree k or k-*neighbors*.*

Definition 6 (Successor Graph) *The* successor graph *G_C of C consists of all successor configurations of C. In it, neighboring configurations are connected by an edge. A successor graph where all neighbors of degree up to k are connected by an edge is called a k-*successor graph*.*

Lemma 1 (Algorithm Reduction) *For every algorithm $\mathcal{A}$ which has to achieve a goal by delivering some value out of a set of possible decision values $\mathcal{V}$ with $|\mathcal{V}| \geq 2$ as the result of its execution, and which has configurations C where deciding at once is possible and results in decision $v \in \mathcal{V}$, but allowing the algorithm to take more rounds may revise its decision to $v' \in \mathcal{V}$, $v' \neq v$, the algorithm $\mathcal{A}'$ which sticks to the decision made in C in all further rounds will also achieve the goal.*

Proof: For any execution run σ, $\mathcal{A}'$ is the same as $\mathcal{A}$ until the first decision is made. Therefore, if $\mathcal{A}$ achieves its goal, then every σ must come to a decision in a finite number of rounds, and $\mathcal{A}'$ too must be able to decide in every σ in a finite number of rounds. Since the first decision $\mathcal{A}$ makes is already a valid decision achieving the goal, $\mathcal{A}'$ will achieve the goal as well. □

For our impossibility proofs, we will only consider algorithms with irreversible decisions and will show that no such algorithm exists which can solve the goal, since in this case no algorithm can exist: If an algorithm with reversible decisions did exist, we could derive an irreversible algorithm from it which would also achieve the goal.

Definition 7 (Bivalent) *A configuration C is* bivalent *if there is both a finite execution run σ_v which, when applied to C, results in the decision $v \in \mathcal{V}$, and a finite execution run $\sigma_{v'}$ which, when applied to C, results in the decision $v' \in \mathcal{V}$, $v' \neq v$. A configuration that is not bivalent is called* univalent*. If the decision value of a univalent configuration is v, then the configuration is also called* v-valent*.*

Let us assume a fully connected synchronous point-to-point system with n processes. We will show that if the system fulfills the conditions

(I1) the k-successor graph is connected for some $1 \leq k < n$, and

(I2) for any two configurations connected in the k-successor graph, we can put the affected processes[5] into a set $\mathcal{P}_r$ with $k \leq |\mathcal{P}_r| < n$ which has the property that there exists an (infinite) execution run σ in which the processes in $\mathcal{P}_r$ only communicate among themselves;
at the start of the algorithm, we can put any process into $\mathcal{P}_r$,

then there is no deterministic algorithm solving consensus in this system.

Lemma 2 (Equal Valence) *In a system fulfilling the conditions (I1) and (I2), if all successor configurations of some configuration C of an algorithm that claims to solve consensus are univalent, then they must all have the same valence.*

Proof: Assume that all successor configurations are univalent. Let us consider two configurations C' and C'' which are neighbors with degree of at most k. Then at most $k < n$ processes are affected by differing message arrivals/losses. According to (I2) we can put these processes into a set $\mathcal{P}_r$ with $k \leq |\mathcal{P}_r| < n$ of processes which may only communicate among themselves. Consider the (infinite) execution run σ where the processes in $\mathcal{P}_r$ only communicate among themselves. If we apply σ to C' and if we assume that C' is v-valent, then the algorithm must decide on v during run σ. However, for the processes not in $\mathcal{P}_r$, there is no difference between the configurations resulting from applying σ to C' and from applying it to C'', so they must decide on v in C'' as well. Since C'' is univalent, there can only be one decision value for all possible execution runs and C'' must be v-valent as well.

The above argument can be extended to transitive relations as well: If C and D are k-neighbors and D and E are k-neighbors, then C and E must have the same valence. Since by (I1) the successor graph is connected, any two configurations are indirectly

[5] That is, the processes which have experienced a message loss in one of the configurations but not in the other.

connected over a path of k-neighbors and therefore all configurations must have the same valence. □

The definition of bivalence and the above proof make it clear why we only want to consider irreversible algorithms: Since the proof relies on the fact that the algorithm does not change its mind once it has decided, we must ensure that there is no algorithm which may decide on two different values at different stages of the same execution run. By our definition, this would be a bivalent configuration, but the algorithm would still achieve its goal in a finite number of rounds, invalidating our proof.

Lemma 3 (Initial Configuration) *In a system fulfilling the conditions (I1) and (I2), there exists a bivalent initial configuration for all* $n \geq 2$.

Proof: Assume that all initial configurations are univalent, i.e., every execution run from a particular configuration will result in the same decision. Since validity requires that if all processes start with an input value of v, then v must also be the decision value, there must be at least two configurations C_v resp. $C_{v'}$ which decide on v resp. v', $v \neq v'$, and which only differ in the initial value of one process p. From (I2) we know that we can select a set $\mathcal{P}_r$ of $k \geq 1$ processes which only send messages to themselves. Let $p \in \mathcal{P}_r$. Then the configurations C_v and $C_{v'}$ look the same to all processes not in $\mathcal{P}_r$ and they would have to make the same decision in both configurations, which is a contradiction to our assumption. □

Theorem 3 (Impossibility of Consensus) *There is no deterministic algorithm solving consensus in a system fulfilling (I1) and (I2).*

Proof: Assume that we have an algorithm which claims to achieve consensus in this system. We already know from Lemma 3 that there must be a bivalent initial configuration. We will now show that if an algorithm has a bivalent configuration C in round R, then there is at least one bivalent successor configuration in round $R + 1$. Hence, we can conclude by induction that for any round R, there is an execution run up to R which results in a sequence of bivalent configurations in which the algorithm cannot yet decide.

Consider some bivalent configuration C. We want to prove that at least one successor configuration must be bivalent as well. So let us assume that there are only univalent successor configurations. If all configurations are univalent, however, they must all have the same valence according to Lemma 2, so C itself must be univalent, which is a contradiction to our assumption. □

Corollary 1 (Byzantine Agreement) *In a system fulfilling (I1) and (I2), any deterministic algorithm that achieves consensus must achieve Byzantine agreement for the initial value of at least one process* p.

Proof: Consider any deterministic consensus algorithm. Since the algorithm has to terminate, its final configurations must all be univalent, so either all initial configurations are already univalent, or there exist some bivalent configurations which only have univalent successor configurations. With the same argument as used in Lemma 3 for initial configurations, we can show that in this case, there must be two neighboring configurations C_v resp. $C_{v'}$ which decide on v resp. v' and which only differ in the message reception/loss event of one process p. Since this event, which is only known to p, determines the consensus result, all processes effectively agree on the value of p, hence they achieve Byzantine agreement for p. □

Remarks:

1. As we have already indicated, consensus and Byzantine agreement are closely related. In fact, one can be used to implement the other: If we have a consensus algorithm available, we can simply let the transmitter distribute its value and then start the consensus algorithm on every receiver. The initial value of each process is the value obtained from the transmitter. On the other hand, if we have a Byzantine agreement algorithm available, we can start an instance of this algorithm on each process to disseminate its value. For each transmitter, all processes will agree on its initial value, so all processes will obtain the same set of values from the Byzantine agreement instances. The final consensus decision can be derived from this set by using an appropriate function.

   However, Byzantine agreement and consensus are not the same. Although Byzantine agreement can be seen as a consensus algorithm preceded by a value distribution round, the failure model determines what initial configurations the Byzantine agreement consensus (BAC) algorithm must be able to handle. If the transmitter can be arbitrary faulty, then any initial configuration can occur, and the BAC algorithm must have the same properties as a consensus algorithm. If, however, the transmitter is consistent and if there are no link faults, then the BAC algorithm must only be able to achieve consensus for initial configurations where all processes have the same initial value, which requires considerably less work than a consensus algorithm must usually do.

5.2.2 Link Failures

In this section, we will develop a number of theoretical results related to consensus in presence of link failures, see [58] for details. We restrict our attention to a synchronous distributed system of n processes according to Definition 2 here, where only link failures but no process failures are allowed ($f_a = f_s = f_o = f_m = 0$). Our major tool will be a generalization of the well-known Theorem 1 [27], which reveals the importance of unimpaired *bidirectional* communication for solving consensus. It allows to derive (tight) lower bounds on the number of nodes required for solving consensus, both for the case of pure omission link failures and for arbitrary ones.

We start with the well-known proof of Gray's Theorem 1 in the formalization of [37, Thm. 5.1], which uses the following argument: Suppose that the failure-free execution $\overline{E}$ of a two-process system with omission faulty links terminates at the end of round r when starting with initial values $[1, 1]$. By validity, the common decision value must be 1 in $\overline{E}$. Since decisions are irreversible, however, we can safely drop all the messages sent in rounds $> r$ without changing the decision values. The resulting "truncated" execution E shown in Figure 5.4 is obviously feasible.

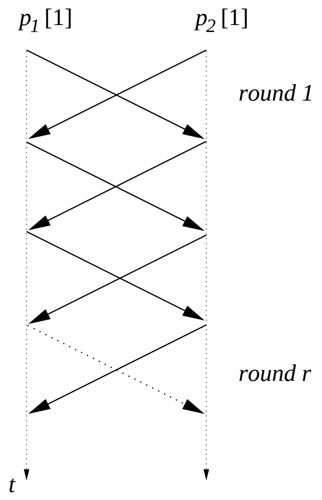

Figure 5.4. *Execution of a two-process synchronous consensus algorithm with link omission failures, starting with initial values* $[1, 1]$*, truncated after round* r *by which both processes decide.*

If we now drop the last (dotted) round r-message from $p_1 \to p_2$ in E, the resulting execution E' is indistinguishable for p_1, so p_1 and, by agreement, also p_2 must eventually decide on 1 as well. Note carefully, however, that p_2 could decide later now, i.e., in some round $r' > r$, in E' — we can only guarantee that the decision value is the same. An analogous argument reveals that we can also drop the round r-message from $p_2 \to p_1$ in E' without changing the decision value. Since p_2 could decide in round $r' > r$ here, the initial "truncation" made it impossible for p_1 to tell p_2 about the lost message in some later round $r + 1, \ldots, r'$.

The above procedure can be repeated until all the messages in all rounds have been dropped, without changing the decision value. Since the processes are now fully isolated from each other in the final execution, changing the initial values to $[1, 0]$ and then to $[0, 0]$ cannot affect the decision value either, but now the outcome of the final execution would violate validity.

As our first result, we will now show that solving consensus is even impossible when a link—viewed as a pair of unidirectional links—loses or, in case of arbitrary link failures, corrupts messages only in one direction, i.e., when either node (but not necessarily both) can withhold its information for an arbitrary number of rounds. Eventually bidirectional communication is hence mandatory for any deterministic consensus algorithm. Solutions exist, however, if the direction of the message loss is fixed, see the remarks following Theorem 4 below.

Unfortunately, this stronger result cannot be shown by generalizing the proof of Theorem 1: We may not simply drop all messages in later rounds to "hide" the effect of dropping/restoring round r-messages here, since this would amount to a link failure in both directions and hence an infeasible execution. Our alternative proof, which could also be employed to show Theorem 1, uses ideas borrowed from the well-known proof of the impossibility of consensus in asynchronous systems [20], as we have mentioned in Section 5.2.1.

Theorem 4 (Unidirectional 2-Process Impossibility) *There is no deterministic algorithm that solves consensus in a synchronous system with two non-faulty processes connected by a lossy link if communication is reliable in only one direction (that may change arbitrarily).*

Proof: We will first show (I1), i.e., that the successor graph is connected: In a 2-process system with one lossy link, the successor graph only consists of the configuration C_{11}— the configuration where both messages arrive— and the two configurations where one of the messages does not arrive. Obviously, this graph is connected.

Property (I2) is equally easy to prove: Since the direction of the link failure may change arbitrarily, we can always attribute the link failure to just one node, thus separating the affected node from its peer.

Since our 2-process system possesses both (I1) and (I2), Theorem 3 implies that consensus is not possible. □

Remarks:

1. If message losses can only occur in one and the same direction, and if that direction is known to the algorithm, then there is a trivial 1-round algorithm that solves consensus in a 2-process system: The process that can communicate with its peer sends its own value and decides upon it; the other process decides upon the value received from its peer.

2. If message losses can only occur in one and the same (but unknown) direction, there is a 2-round algorithm that solves consensus in a 2-process system. In the first round, the initial values v_1, v_2 are exchanged. In the second round, Acks/Nacks signaling successful reception in the former round are sent. The possible message receive patterns for the lucky process, say, p_2, that gets all messages are $(v_1, Ack) \models \min\{v_1, v_2\}$ and $(v_1, Nack) \models v_1$, whereas the less lucky p_1 could get $(v_2, Ack) \models \min\{v_1, v_2\}$, $(v_2, \emptyset) \models \min\{v_1, v_2\}$, $(\emptyset, Ack) \models v_1$, or $(\emptyset, \emptyset) \models v_1$. It is easy to verify that the decision values (given after $\models$ in each case) satisfy validity and agreement.

We now generalize this result to show the following tight lower bound for omission link failures.

Theorem 5 (Lower Bound 1) *Any deterministic algorithm that solves consensus under the system model of Definition 2 for $f_\ell^r = f_\ell^{ro}, f_\ell^s = f_\ell^{so} > 0$ with only omission link failures needs $n > f_\ell^r + f_\ell^s$.*

The proof applies Theorem 3 by proving that a system with $n \leq f_\ell^r + f_\ell^s$ fulfills conditions (I1) and (I2). First, we will show (I2) by demonstrating that any set of f_ℓ^r processes may not send messages to the remaining $n - f_\ell^r$ processes.

Lemma 4 *Any set $\mathcal{V}_r$ of f_ℓ^r processes out of $n \leq f_\ell^r + f_\ell^s$ may only exchange messages among themselves and need not send any messages to processes outside $\mathcal{V}_r$.*

Proof: The proof is trivial. Take any set $\mathcal{V}_r$ of f_ℓ^r processes. Then there are $n - f_\ell^r \leq f_\ell^s$ processes which are not in $\mathcal{V}_r$. Each process in $\mathcal{V}_r$ is allowed to make f_ℓ^s send link faults and thus may fail to send to the at most f_ℓ^s processes not in $\mathcal{V}_r$. Since these processes may experience up to f_ℓ^r receive link faults, they will indeed fail to receive the messages from the processes in $\mathcal{V}_r$. □

Secondly, we will show that the successor graph is connected and thus that (I1) is fulfilled.

Lemma 5 *The successor graph G_C of any configuration C in a system with $f_\ell^r, f_\ell^s > 0$ and $n \leq f_\ell^r + f_\ell^s$ is connected.*

Proof: First, we note that C_{all}, the configuration where all messages to all processes arrive, is in G_C. For every other configuration $C' \in G_C$, a set $\mathcal{M}'$ of at most $n \cdot f_\ell^r$ messages have not arrived. Obviously, the neighboring configuration C'' which is the same as C' except that one message $m \in \mathcal{M}'$ does arrive is also in G_C and has a smaller set $\mathcal{M}'' = \mathcal{M}' \setminus \{m\}$ of lost messages. By restoring the messages one by one, we will always find neighboring configurations which are in G_C as well until we arrive at the configuration C_{all}. □

Remarks:

1. The lower bound $n > f_\ell^r + f_\ell^s$ provided by Theorem 5 is tight, since it is matched by the authenticated algorithm ZA analyzed in Section 5.5.2, see Theorem 11.

2. It is interesting to note that the result of Theorem 5 implies that, in order to be able to solve consensus, any link failure may affect at most a minority of processes only. In the setting of Theorem 1, however, there is no point in considering this case at all: There is no non-empty minority of processes for $n = 2$. We are reasonably convinced that our fairly simple escape from the impossibility result would have been discovered earlier during the past 20 years if Jim Gray had not presented his result for a simple 2-process system.

In order to find a lower bound for arbitrary link failures, we will again start with the special case $f_\ell^s = f_\ell^r = f_\ell^{sa} = f_\ell^{ra} = 1$. The following Theorem 6 shows that no algorithm can solve consensus in a 4-process system in presence of a single arbitrary link failure. This will be proved by means of a technique, which is well-known from showing the impossibility of consensus in a system of 3 Byzantine faulty processes, see e.g. [3, p.104].

Theorem 6 (4-Process Impossibility) *There is no deterministic algorithm that solves consensus under the system model of Definition 2 for a single arbitrary link failure in a 4-process system.*

Proof: We employ a new instance of the "easy impossibility proof techniques" of [19] to show that any deterministic algorithm violates agreement if every process may see an inconsistent value from one of its neighbors. Suppose that our 4 processes execute a distributed algorithm consisting of specific programs A, B, C, D, which solve consensus under the system model of Definition 2 with $f_\ell^{ra} = f_\ell^{sa} = f_\ell^{s} = f_\ell^{r} = 1$. In order to derive a contradiction, we arrange 8 processes in a cube as shown in Figure 5.5. For example, the lower leftmost process labeled $A[0]$ executes program A starting with initial value 0 (the $\boxed{0}$ on its left displays this process's decision value, as explained below). Note carefully that all processes and all links are assumed to be non-faulty here.

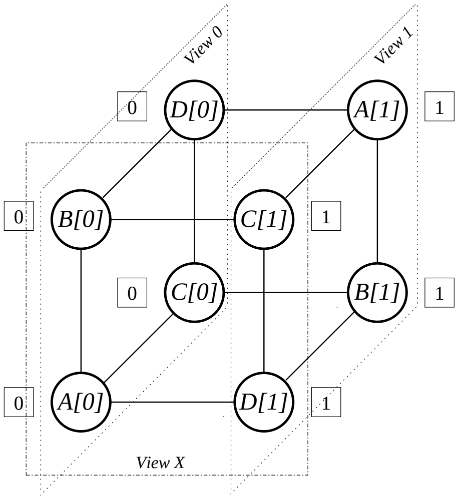

Figure 5.5. *Topology used for proving the violation of agreement in a 4-process system. 8 non-faulty processes with perfect links are arranged in a cube in a neighborhood-preserving way. The assignment of initial values ensures that all processes in view 0 resp. view 1 decide* 0 *resp.* 1*, but this violates agreement in view* X*.*

Of course, dealing with a solution for a 4-process system, we cannot expect to achieve consensus in the 8-process system of Figure 5.5. However, due to the special assignment of programs to processes, each process observes a neighborhood as in a 4-process system. More specifically, the 4 processes at any side of the cube (we call it a *view*) can be interpreted as an instance of a legitimate 4-process system. In fact, as can be checked easily, our assignment ensures that any process in a view is connected to exactly one process outside this view. Since we assumed that every process may see an arbitrary faulty input from at most one neighbor, the input from the process outside the view may be arbitrary—it just appears like an arbitrary faulty process.

Now consider the processes in view 0, which all have initial value 0. By the validity property for consensus, all processes must decide 0 here (the initial value 1 of the processes outside view 0 do not matter, as they are considered arbitrary faulty w.r.t. view 0). Similarly, in view 1, all processes must decide 1 since they have initial value 1. But now the processes in view X face a problem: Since e.g. the lower leftmost process $A[0]$ observes exactly the same messages in view X as in view 0, it must decide 0 as observed above. Similarly, as the lower rightmost process $D[1]$ observes exactly the same messages in view X as in view 1, it must decide 1—but this would violate agreement in the 4-process system corresponding to view X. We hence established the required contradiction, thereby completing the proof of Theorem 6. $\Box$

Again, we generalize this result to show the following tight lower bound for arbitrary link failures.

Theorem 7 (Lower Bound 2) *Any deterministic algorithm that solves consensus under the system model of Definition 2 for $f_\ell^r = f_\ell^{ra}, f_\ell^s = f_\ell^{sa} > 0$ with only arbitrary link failures needs $n > 2f_\ell^r + 2f_\ell^s$.*

Proof: Assume that $n = 2f_\ell^r + 2f_\ell^s$. We partition the set of all processes P into 4 subsets P_A, P_B, P_C, P_D with $|P_A| = |P_B| = f_\ell^s$, $|P_C| = |P_D| = f_\ell^r$. From Corollary 1 we know that there must be at least one process p for which we have to achieve Byzantine agreement. Let us first assume that p holds the initial value $v_p = v$, which it distributes to all processes. At most f_ℓ^s of these may receive a different value v' instead, which we will combine into the set P_A. All remaining processes obtain the correct value v, which is also the desired consensus result. We will call this configuration C_v. If, however, p holds the initial value $v_p = v' \neq v$, which is received incorrectly as v by the f_ℓ^s processes of P_B, whereas all other processes obtain the correct value v', we obtain the configuration $C_{v'}$. In this configuration, $v_p = v'$ is the desired decision value. Obviously, the processes in P_A and P_B hold the same values in both C_v and $C_{v'}$, so for them the two configurations are indistinguishable and they need at least one more round to decide.

Starting with C_v, our failure model allows us to generate a feasible message exchange pattern resulting in configuration C'_v in which each process in P_A receives the faulty value v' from every process in P_D, since any process may fail to receive the correct value from at most $f_\ell^r = |P_D|$ senders, whereas any process may produce up to $f_\ell^s = |P_A|$ send link failures. Similarly, the processes in P_B may obtain the faulty value v' from all processes in P_C. Since each process $p \in P_A \cup P_B$ also receives f_ℓ^s values v' from the processes in P_A and f_ℓ^s values v from the processes in P_B, p obtains $f_\ell^s + f_\ell^r$ values v and $f_\ell^s + f_\ell^r$ values v'. Each process $q \in P_C \cup P_D$, on the other hand, receives a majority of $f_\ell^s + 2f_\ell^r$ values v and can decide on v. With a similar message exchange pattern, we can transform $C_{v'}$ into configuration $C'_{v'}$ where each process $p \in P_A \cup P_B$ holds $f_\ell^s + f_\ell^r$ values v and $f_\ell^s + f_\ell^r$ values v', whereas the processes in $P_C \cup P_D$ hold a majority of $f_\ell^s + 2f_\ell^r$ values v' and can decide on v'. Clearly, configurations C'_v and $C'_{v'}$ are still indistinguishable for all processes in $P_A \cup P_B$.

If we add any more rounds, however, the situation does not improve: With the same message exchange patterns as before, we can cause all processes in $P_A \cup P_B$ to receive equally many values v and v', whereas the processes in $P_C \cup P_D$ obtain a majority of correct values. Hence, we can generate an infinite execution run in which the processes in $P_A \cup P_B$ can never decide. □

Remarks:

1. The lower bound $n > f_\ell^r + f_\ell^s$ provided by Theorem 7 is tight, since it is matched by the unauthenticated algorithm OMH analyzed in Section 5.3.1, see Theorem 8.

5.2.3 Process Failures

Since our failure model of Definition 2 allows for dynamic link failures, it would be tempting to try to model reality even more by assuming that process failures are dynamic as well. However, it is easy to see that in such a case, consensus is not solvable anymore, as we will demonstrate in Lemma 6.

Lemma 6 *There is no deterministic algorithm solving consensus in a synchronous system of n fully connected processes if there is one omission faulty process which may change in every round.*

Proof: To prove the lemma, we will first show that the successor graph G_C of any configuration C in a system of n fully connected processes and one dynamic omission fault per round is connected.

The proof uses nearly the same arguments as the one of Lemma 5: Again, we note that C_{all}, the configuration where all messages to all processes arrive, is in G_C. For every other configuration $C' \in G_C$, a set $\mathcal{M}'$ of at most $n-1$ messages has not arrived. Obviously, the neighboring configuration C'' which is the same as C' except that one message $m \in \mathcal{M}'$ does arrive is also in G_C and has a smaller set $\mathcal{M}'' = \mathcal{M}' \setminus \{m\}$ of lost messages. By restoring the messages one by one, we will again find neighboring configurations which are in G_C as well until we arrive at the configuration C_{all}. Hence, property (I2) holds.

Since the omission fault can be attributed to any process p, we can always set $\mathcal{P}_r = \{p\}$ (I1). With these two properties, we can apply Theorem 3 to conclude that consensus is not possible. □

5.3 Hybrid Oral Messages Algorithms

In this section, we will show that the *Hybrid Oral Messages* algorithm OMH derived from [34] in [35], as well as a uniform variant OMHU, solves Byzantine agreement under the system model of Section 5.1.

5.3.1 OMH

The algorithm OMH as specified in Definition 8 below uses two primitives:

- The *wrapper function* $R(v)$ encodes a statement "I am reporting v" as a unique value. Reporting is undone by means of the inverse function $R^{-1}(v)$, which must guarantee $R^{-1}(R(v)) = v$. Note that only $\emptyset$, $R(\emptyset)$, $R(R(\emptyset))$, $R(R(R(\emptyset)))$, ... must actually be distinguishable here; for each legitimate value v, we can allow $R(v) = R^{-1}(v) = v$.

- The *hybrid-majority* of a set $\mathcal{V}$ of values provides the majority of all non-$\emptyset$ values in $\mathcal{V}$. If no majority exists, the default value $R(\emptyset)$ is returned. Note that this particular default value is required for securing validity in presence of an omission faulty transmitter, see the proof of Lemma 8.

Consult [35] for a detailed discussion of the above primitives and the operation of OMH in general.

Definition 8 (Algorithm OMH [35]) *The Hybrid Oral Message algorithm OMH is defined recursively as follows:*

OMH(0):

1. *The transmitter t sends its value v to every receiver and delivers $v_t = v$.*

2. *Every receiver p delivers the value v_p received from the transmitter, or the value $\emptyset$ if a missing or manifestly erroneous value was received.*

OMH(m), $m > 0$:

1. *The transmitter t sends its value v to every receiver and delivers $v_t = v$.*

2. *For every p, let w_p be the value receiver p receives from the transmitter, or $\emptyset$ if no value or a manifestly bad value was received.*

   Every receiver p acts as the transmitter in algorithm OMH($m-1$) to communicate the value $R(w_p)$ to all receivers.

3. *For every p and $q \neq p$, let w_p^q be the value receiver p delivers as the result of OMH($m-1$) initiated by receiver q in step 2 above. Every receiver p calculates the hybrid-majority value of all values w_p^q and its own value $w_p^p = R(w_p)$, and applies R^{-1} to that value. This result is delivered as v_p.*

Remarks:

1. There are $n-1$ receivers in the first instance OMH(m) of the algorithm; the transmitter does not participate in any way in further recursive instances. Our n-node, $m+1$-round Byzantine agreement algorithm OMH(m) can hence be viewed as an initial broadcast of the transmitter's value to all receivers combined with an $n-1$-node, m-round consensus algorithm.

2. In our failure model, the transmitter's delivery $v_t = v$ must be considered as the result of sending its value to itself: In case of a (strong) manifest fault, the value delivered by the transmitter must be $\emptyset$ as well. Recall also that both (A1^s) and (A1^r) are valid for any set $\mathcal{R}$ resp. $\mathcal{S}$ of receiving resp. sending processes, including sender resp. receiver. Hence, there is no need to treat the case $q \neq p$ and $q = p$ differently in step 3 in OMH(1). We will use this fact in order to immediately apply Lemma 7 given below in OMH's analysis.

3. During the recursive execution of OMH, multiple (in fact, quite many) instances of OMH are concurrently active[6] in each round. In OMH's description, we did not explicitly address the question of how received messages are assigned to the unique recursive instance they belong to. The unique *id* of the transmitter process is appended to each message for this purpose, which produces a string of ids $p_1 p_2 \dots p_k$ that uniquely determines the particular recursive instance (and, by its complement, the set of participating receivers). This string must be reconstructed upon reception of a $\emptyset$-value and included in the $R(\emptyset)$-message prior to submitting it to further recursive instances.

4. By expanding OMH's recursive description, it is easy to see that the execution of OMH(m) consists of $m + 1$ rounds where messages are wrapped and forwarded only, followed by the backwards-recursive computation (hybrid-majority + unwrapping) in step 3 of all instances, cp. the EIG-tree representation in [3, p. 105]. Consequently, not only non-faulty but also obedient processes can always compute the correct value to be forwarded in each particular instance—and hence in every round—independently of their past behavior, cp. Remark 8 on Definition 2.

By adopting and extending the analysis of [35] to our perception-based failure model, we will now show that OMH satisfies (B1) and (B2).

We start our derivations with a widely applicable technical Lemma 7, which shows that a common decision value is reached among all obedient processes if sufficiently many processes sent the same value. More specifically, let us assume that all processes $p \in \mathcal{S}$ of some set of processes $\mathcal{S}$ *convey* their local values w_p to an arbitrary obedient receiver q (possibly $q \in \mathcal{S}$) in some round of the execution of a certain (multi-round) algorithm. Conveying means that all $p \in \mathcal{S}$ send $R(w_p)$ to q, which computes R^{-1} of the hybrid-majority among the set of received values as its result v_q. Conveying occurs in steps 2 and 3 of OMH(1), for example, recall Remark 2 on Definition 8. Note carefully that a process p_m on a manifest faulty sender node may very well send its $R(w_{p_m})$ in the conveying round, since the manifest fault may occur in some other round or process.

Lemma 7 *For any $f, f_a, f_o, f_s, f_m, f_\ell^s, f_\ell^r, f_\ell^{ra} \geq 0$, let some set of processes $\mathcal{S}$ with*

(a) $|\mathcal{S}| > 2f + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + 2f_o + f_m$ *resp.*

[6] Note that all the peer processes belonging to a single concurrent instance are considered a separate system of n processes.

(b) $|\mathcal{S}| > 2f + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + f_o + f_m$

be given. All processes in $\mathcal{S}$ are assumed to convey their values to an obedient receiver q in some round of the execution of a distributed algorithm. If all but at most f of the (a) non-faulty or manifest faulty resp. (b) obedient processes $p \in \mathcal{S}$

(1) own the same value $w_p = \nu$, then q computes $v_q = \nu$,

(2) own values taken from a set $\mathcal{W}$ with $|\mathcal{W}| \geq 2$, then $v_q \in \mathcal{W} \cup \{\emptyset\}$.

Proof: Let $n' = |\mathcal{S}|$, $\mathcal{S}_f \subseteq \mathcal{S}$ be the set of the at most f exempted processes, and f_o' resp. f_m' be the actual number[7] of omission resp. manifest failures in $\mathcal{S}$ that affect our receiver q in the conveying round by a $\emptyset$-value (or, in case of an additional arbitrary link failure, even a faulty value).

In addition, let ${f_{\ell n}^r}' \leq f_\ell^r$ be the number of actual receive link failures at q that hit non-faulty senders $\in \mathcal{S} \backslash \mathcal{S}_f$, ${f_{\ell m}^{ra}}'$ resp. ${f_{\ell o}^{ra}}'$ be the actual number of arbitrary receive link failures at q hitting manifest resp. omission faulty senders, and ${f_\ell^{ro}}''$ be the number of actual receive link omission failures at q hitting all but actually manifest or omission faulty senders. Clearly, we have $f_\ell^{ra} + {f_\ell^{ro}}'' \geq {f_{\ell n}^r}' + {f_{\ell o}^{ra}}' + {f_{\ell m}^{ra}}'$, since we can split ${f_{\ell n}^r}'$ into its arbitrary and omissive components ${f_{\ell n}^r}' = {f_{\ell n}^{ra}}' + {f_{\ell n}^{ro}}'$ and $f_\ell^{ra} \geq {f_{\ell n}^{ra}}' + {f_{\ell o}^{ra}}' + {f_{\ell m}^{ra}}'$ and ${f_\ell^{ro}}'' \geq {f_{\ell n}^{ro}}'$.

We start with proving item (1) of the statements-part of our lemma. In case (a), where all non-faulty and manifest faulty processes $\in \mathcal{S} \backslash \mathcal{S}_f$ own the same value $R(\nu)$, our obedient receiver q will obtain at least

$$n_q' = n' - f - f_a - f_s - f_o - f_m' - {f_{\ell n}^r}'$$

identical values $R(\nu)$. It will also get at most

$$n_q'' = n' - f_o' - f_m' - {f_\ell^{ro}}'' + {f_{\ell o}^{ra}}' + {f_{\ell m}^{ra}}'$$

non-$\emptyset$ values. Since $n' > 2f + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + 2f_o + f_m$ here, we have

$$\begin{aligned} 2n_q' - n_q'' &= n' - 2f - 2(f_a + f_s) - 2f_o - f_m' - 2{f_{\ell n}^r}' \\ &\quad + f_o' + {f_\ell^{ro}}'' - {f_{\ell o}^{ra}}' - {f_{\ell m}^{ra}}' \\ &> (f_m - f_m') + f_o' + (f_\ell^r - {f_{\ell n}^r}') \\ &\quad + (f_\ell^{ra} + {f_\ell^{ro}}'' - {f_{\ell n}^r}' - {f_{\ell o}^{ra}}' - {f_{\ell m}^{ra}}') \\ &\geq 0. \end{aligned} \tag{5.2}$$

Therefore, $R(\nu)$ will indeed win the hybrid majority at process q.

In case (b), where all obedient processes $\in \mathcal{S} \backslash \mathcal{S}_f$ own the same value $R(\nu)$, q will obtain at least

$$\overline{n}_q' = n' - f - f_a - f_s - f_o' - f_m' - {f_{\ell n}^r}'$$

[7]In this and subsequent proofs, we often need the actual number of failures of a given type. It will be denoted by priming the bound on the maximum number of failures: $0 \leq f_o' \leq f_o$ etc.

identical values $R(\nu)$ and will get at most

$$\overline{n}''_q = n' - f'_o - f'_m - f_\ell^{ro\prime\prime} + f_{\ell o}^{ra\prime} + f_{\ell m}^{ra\,\prime}$$

non-$\emptyset$ values. Since $n' > 2f + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + f_o + f_m$ here, we get

$$\begin{aligned} 2\overline{n}'_q - \overline{n}''_q &= n' - 2f - 2f_a - 2f_s - f'_o - f'_m - \\ &\quad -2f_{\ell n}^{r\,\prime} + f_\ell^{ro\prime\prime} - f_{\ell o}^{ra\prime} - f_{\ell m}^{ra\,\prime} \\ &> (f_o - f'_o) + (f_m - f'_m) + (f_\ell^r - f_{\ell n}^{r\,\prime}) \\ &\quad +(f_\ell^{ra} + f_\ell^{ro\prime\prime} - f_{\ell n}^{r\,\prime} - f_{\ell o}^{ra\prime} - f_{\ell m}^{ra\,\prime}) \\ &\geq 0, \end{aligned} \tag{5.3}$$

by the same reasoning as above, so $R(\nu)$ will again win the hybrid-majority at q.

As far as part (2) of the statements of Lemma 7 is concerned, if the appropriate processes $\in \mathcal{S}\backslash\mathcal{S}_f$ did not all send the same value but rather values from a set $\mathcal{W}$, then we cannot always guarantee that receiver q obtains a majority for any of these values (although this could happen). However, since (5.2) resp. (5.3) holds if we temporally consider each value from $\mathcal{W}$ a single legitimate value X, it turns out that there cannot be a majority for any value $\notin \mathcal{W}$. Hence, only the default value $R(\emptyset)$ can be returned by the hybrid-majority primitive here, which leads to the alternative return value $v_q = \emptyset$. $\square$

Now we are ready for proving that OMH satisfies the validity property (B2). Note that Lemma 8 is void in case of an arbitrary faulty transmitter, since (B2) does not say anything about the value a receiver ascribes to the transmitter in this case.

Lemma 8 (Validity OMH) *For any $m \geq \min\{1, f_\ell^s\}$ and any f_a, f_s, f_o, f_m, f_ℓ^s, f_ℓ^r, $f_\ell^{ra} \geq 0$, the algorithm OMH(m) satisfies the validity property (B2) if there are strictly more than $2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + f_o + f_m + m$ participating processes.*

Proof: Proving (B2) amounts to showing that any obedient[8] process p, including the transmitter, delivers

(1) $v_p = \nu = v$, if the transmitter is non-faulty,
(2) $v_p = \nu = \emptyset$, if the transmitter is manifest faulty,
(3) $v_p = v$ or $v_p = \emptyset$, if the transmitter is omission faulty,
(4) $v_p = \nu$, if the transmitter is symmetric faulty.

The actual proof is by induction on m.

If there are no link failures, i.e., if $f_\ell^s = 0$, induction will start at $m = 0$: In OMH(0), the transmitter sends its value ν ($\nu = v$ if the transmitter is non-faulty) to all receivers

[8]In order to show (B2), it would be sufficient to replace obedient by non-faulty here. However, our proof shows an even stronger result: OMH actually satisfies uniform validity (UB2) as introduced in Section 5.3.2.

and itself; the received values are simply delivered as v_p. Properties (1)–(4) follow immediately from the definition of process failures in Definition 2.

If $f_\ell^s > 0$, however, induction must start at $m = 1$ as the base case: According to the definition of OMH(1), every receiver q of step 1 of OMH(1) uses OMH(0) to disseminate its w_q to all receivers p (including itself), which in turn compute the majority of the non-$\emptyset$ values delivered in OMH(0) to obtain the result v_p of OMH(1). Let us first consider the cases where the transmitter is consistent, i.e., (1), (2) and (4) above: Abbreviating the number of initially participating receivers by

$$n' \geq 2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2f_a + 2f_s + f_o + f_m + 1,$$

all but at most f_ℓ^s of the obedient receivers p of step 1 of OMH(1) get the same $w_p = \nu$ (we can simply assume $\nu = \emptyset$ in case of a manifest faulty transmitter) due to the transmitter's at most f_ℓ^s link failures according to (A1^s). Therefore, we can apply item (1), case (b) of Lemma 7 with $f = f_\ell^s$ to conclude that every obedient receiver will deliver the same value $w_p = \nu$ as the result of OMH(1). In the remaining case (3), where the transmitter is omission faulty, all obedient receivers get either $w_p = \nu$ or $w_p = \emptyset$. In this case, item (2), case (b) of Lemma 7 with $f = f_\ell^s$ and $\mathcal{W} = \{\nu, \emptyset\}$ shows that every obedient receiver q must deliver either ν or $\emptyset$ as required.

Assuming that the lemma is already true for $m - 1 \geq \min\{1, f_\ell^s\}$, we will show that it is also true for m: For a consistent transmitter, we have at least $n' - f_\ell^s - f_a - f_s$ obedient receivers q of step 1 of OMH(m) that apply OMH($m-1$) to disseminate their $R(w_q) = R(\nu)$. Since both m and the number of participants decreased by one, we can apply the induction hypothesis to OMH($m-1$) to conclude that every obedient receiver p actually delivers $R(\nu)$ in this step for every non-faulty transmitter q. Consequently, any obedient receiver p must have at least

$$\overline{n}'_p = n' - f_\ell^s - f_a - f_s - f'_o - f'_m$$

values equal to $R(\nu)$ among the at most $\overline{n}''_p = n' - f'_o - f'_m$ non-$\emptyset$ values it may have got at all. Herein, $f'_m \leq f_m$ resp. $f'_o \leq f_o$ gives the number of $\emptyset$-values delivered to receiver p by OMH($m - 1$) for manifest resp. omission faulty transmitters q. Since $m > 0$,

$$\begin{aligned} 2\overline{n}'_p - \overline{n}''_p &= n' - 2f_\ell^s - 2f_a - 2f_s - f'_o - f'_m \\ &\geq f_\ell^r + f_\ell^{ra} + (f_o - f'_o) + (f_m - f'_m) + m \\ &> 0, \end{aligned} \tag{5.4}$$

so $R(\nu)$ wins the hybrid-majority at any obedient process p and the final delivery value $v_p = \nu = R^{-1}(R(\nu))$ follows.

For the remaining case (3), exactly the same reasoning as in the proof of item (2) of Lemma 7 reveals that only the default value $R(\emptyset)$ can be returned by hybrid-majority if no majority of either $R(\nu)$ or $R(\emptyset)$ exists. This eventually completes the proof of Lemma 8. □

With the help of Lemma 8, it is not too difficult to show the major Theorem 8.

Theorem 8 (Agreement and Validity OMH) *For any $m \geq f_a + f_o + \min\{1, f_\ell^s\}$ and f_a, f_o, f_s, f_m, f_ℓ^s, f_ℓ^r, $f_\ell^{ra} \geq 0$, the algorithm OMH(m) satisfies agreement (B1) and validity (B2) if there are strictly more than $2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + f_o + f_m + m$ participating processes.*

Proof: Since Lemma 8 is applicable under the conditions of Theorem 8, validity (B2) is evident. Hence, we need to show agreement (B1) only.

The proof is by induction on m and is an extension of the one of [35]. In the base case $m = \min\{1, f_\ell^s\}$, we must have $f_a = f_o = 0$ since $m \geq f_a + f_o + \min\{1, f_\ell^s\}$ by assumption. Hence, the transmitter must not be arbitrary or omission faulty here, such that Lemma 8 also implies (B1).

We can therefore assume that (B1) is satisfied by OMH($m-1$) with $m - 1 \geq \min\{1, f_\ell^s\}$, and have to prove this for OMH(m). Again, it suffices to consider the case where the transmitter is arbitrary or omission faulty, since Lemma 8 also implies (B1) in the other cases. Since we have at most $f_a + f_o \leq m - \min\{1, f_\ell^s\} \geq 1$ arbitrary or omission faulty processes and the transmitter is one of those, either (1) at most $f_a - 1$ arbitrary faulty processes or (2) at most $f_o - 1$ omission faulty ones remain among the strictly more than $2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + f_o + f_m + m - 1$ processes. Since obviously

$$\begin{aligned}2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + f_o + f_m + m - 1 \ > \\ 2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2([f_a - 1] + f_s) + f_o + f_m + [m - 1]\end{aligned}$$

as well as

$$\begin{aligned}2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + f_o + f_m + m - 1 \ > \\ 2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + [f_o - 1] + f_m + [m - 1],\end{aligned}$$

we can apply the induction hypothesis and conclude that any OMH($m-1$) satisfies (B1). Hence, for any q, any two non-faulty processes deliver the same value for w_p^q in step (3). Note carefully that this actually follows from (B2) if one of the two receivers is process q, and from (B1) otherwise. Consequently, any two non-faulty receivers get the same vector of values and hence the same hybrid-majority, which eventually proves (B1) for OMH(m). □

Remarks:

1. The proof for agreement uses only the validity property and the fact that f_a and f_o are added to the number of rounds required by validity. Hence, every consensus algorithm of this type that achieves validity for a given m will also achieve agreement for $m' = m + f_a + f_o$.

2. Lemma 8 and Theorem 8 and their proofs are also valid if the at most f_m manifest faults were not strong but rather weak manifest faults, cf. Remark 6 on Definition 2. Since a weak manifest fault allows the manifest faulty sender process to receive the correct value instead of $\emptyset$, it is usually harder to tolerate than a strong

manifest one. This is not the case here, however, since properties (B1) and (B2) need only hold for non-faulty but not manifest faulty processes — self-delivery at a non-faulty node is always correct.

3. The proof of Lemma 8 revealed that OMH satisfies *uniform* validity (UB2) as introduced in Section 5.3.2, recall Footnote 8: Uniform validity holds not only at non-faulty but also at (alive) manifest and omission faulty processes. By using this fact in the proof of Theorem 8, it is easy to show that OMH provides agreement not only at non-faulty but also at strong manifest faulty processes: The induction step (last paragraph in the proof of Theorem 8) is valid when one of the two receivers is the manifest faulty transmitter process q here as well.

4. It is apparent from Theorem 8 that $2f_o$ processes are required by OMH to tolerate f_o omission faulty processes, which is definitely sub-optimal: Algorithms like the one of [45] require only $f_o < n - 1$. This is not due to an overly conservative analysis, but rather the price paid for OMH's ability to mask additional symmetric and arbitrary failures.

5. From the proof of Lemma 8, it is apparent that $(A1^r)$—and one additional round—is only required to eventually rule out the inconsistencies caused by $(A1^s)$. This is solely done in the base case $m = 1$ of the induction, which implies that limiting the number of link failures for a single receiver by f_ℓ^r according to $(A1^r)$ is only required in the last round, where in turn $(A1^s)$ is not explicitly used. This supports our approach of considering both types of failures independently from each other, recall Remark 2 on Definition 2.

6. For OMH, receive link failures $(A1^r)$ resulting in an omission are easier to tolerate than those that produce a value failure, cf. Theorem 8, since $f_\ell^r + f_\ell^{ra} = f_\ell^{ro} + 2f_\ell^{ra}$ with f_ℓ^{ro} bounding the number of "pure" omission failures. This is not the case for broadcast link failures $(A1^s)$, since an omission failure appears like a value failure in all but the last round (where send link failures do not matter, recall Remark 5 above) due to wrapping all $\emptyset$-values with $R()$.

5.3.2 Uniform Byzantine Agreement

Properties (B1) and (B2) in Section 5.3.1 require agreement and validity to hold only on non-faulty processes. Since there are applications where the behavior of benign faulty processes also matters, cp. [59, Sec. 4], one could ask whether it is possible to extend those properties to obedient processes. In this section, we will therefore consider the following *uniform* properties [29]:

(UB1) (*Uniform Agreement*): If processes p and q are both obedient, then both deliver the same $v_p = v_q$.

(UB2) (*Uniform Validity*): If process p is obedient, the value v_p delivered by p is

- v, if the transmitter is non-faulty,

- $\emptyset$, if the transmitter is manifest faulty,
- v or $\emptyset$, if the transmitter is omission faulty,
- the value actually sent, if the transmitter is symmetric faulty,
- unspecified, if the transmitter is arbitrary faulty.

Note that this definition does not extend (B1) and (B2) to symmetric and arbitrary faulty processes, since this would contradict the impossibility of uniform consensus in presence of Byzantine failures postulated in [10]. Nevertheless, if symmetric and arbitrary faulty processes would faithfully execute the algorithm, i.e., would produce their faults only when emitting messages, then our algorithm below would achieve uniform consensus for them as well.

Originally, we (erroneously) conjectured that the proofs of Lemma 8 and Theorem 8 carry over literally to uniform validity (UB2) and uniform agreement (UB1). Formal verification in [51] showed that this is indeed true for (UB2), but fails in case of (UB1): OMH does not satisfy (UB1) due to the "asymmetry" of transmitter and receivers in conjunction with the ambiguity of (UB2) for omission faulty transmitters. A simple counterexample is an omission faulty transmitter that sends v to itself but $\emptyset$ to all other processes. The receivers will all agree on $\emptyset$ but the transmitter will deliver v. This problem causes the induction step in the proof of Theorem 8 (last paragraph) to fail in the case where one of the two receivers is the transmitter process q, since (B2) does not guarantee agreement if q is omission faulty. To guarantee (UB1), OMH must hence be modified.

Rushby [51] proposed a symmetric version of OMH, where the transmitter participates in recursive instances as well. This algorithm guarantees uniform validity and agreement for weak manifest faulty processes, but not for omission faulty processes.

A straightforward algorithm, which guarantees (UB1) and (UB2) without restrictions, could be built atop of OMH by letting all receivers convey their delivered value back to the transmitter of each recursive instance; the transmitter would then deliver the unwrapped result of the hybrid majority among the received values and its own value. However, this variant would almost double the number of rounds and messages required.

This overhead is avoided by the following simple uniform algorithm OMHU, which employs just a single additional round with a full message exchange: In the final round, all processes (including the transmitter) convey the wrapped value delivered locally by the original OMH to each other. The final decision value is then computed by unwrapping the result of a hybrid majority vote applied to the received values. Lemma 9 and Theorem 9 below show that OMHU satisfy uniform agreement (UB1) and uniform validity (UB2).

Lemma 9 (Validity OMHU) *For any $m \geq \min\{1, f_\ell^s\}$ and any f_a, f_o, f_s, f_m, f_ℓ^s, f_ℓ^r, $f_\ell^{ra} \geq 0$, the uniform algorithm OMHU(m) satisfies uniform validity (UB2) if there are strictly more than $2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + f_o + f_m + m$ participating processes.*

Proof: From the proof of Lemma 8, we know that, in case of a consistent transmitter, all obedient processes in OMH deliver the same value ν that is conveyed to all processes in the additional round of OMHU. Since at least $2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2f_a + 2f_s + f_o + f_m + \min\{1, f_\ell^s\} + 1$ processes participate here, we can apply item (1), case (a) of Lemma 7 with $f = 0$ to conclude that every obedient process will receive enough values to eventually deliver ν.

In case of an omission faulty transmitter, the obedient processes in OMH need not agree upon the same value at the end of OMH but may deliver either ν or $\emptyset$. However, item (2), case (a) of Lemma 7 with $f = 0$ and $\mathcal{W} = \{\nu, \emptyset\}$ applies here. Since $\mathcal{W} \cup \{\emptyset\} = \mathcal{W}$, any obedient process can deliver ν or $\emptyset$ only. □

Theorem 9 (Validity and Agreement OMHU) *For any $m \geq f_a + f_o + \min\{1, f_\ell^s\}$ and any f_a, f_o, f_s, f_m, f_ℓ^s, f_ℓ^r, $f_\ell^{ra} \geq 0$, the uniform algorithm OMHU(m) satisfies uniform agreement (UB1) and uniform validity (UB2) if there are strictly more than $2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + f_o + f_m + m$ participating processes.*

Proof: Due to Lemma 9, it only remains to prove (UB1). First of all, as noted in Remark 3 on Theorem 8, the original OMH satisfies agreement (B1) not only on non-faulty, but also on (strong) manifest faulty processes. Hence, both non-faulty and (alive) manifest faulty processes in OMH deliver the same value ν. Since there are at least $2f_\ell^s + f_\ell^r + f_\ell^{ra} + 3f_a + 2f_s + 2f_o + f_m + \min\{1, f_\ell^s\} + 1$ participating processes here, item (1), case (a) of Lemma 7 for $f = 0$ amply ensures that all obedient processes will deliver ν as the result of the full message exchange in the final round. □

Comparison with OMH shows that OMHU achieves uniform agreement and validity with the same number of processes but one additional round, where $n(n-1)$ messages are exchanged.

5.4 Authentication

Written messages [34] extend oral messages by assuming that (1) it is impossible to alter a message without being detected at the receiver, and (2) that the originator of a message's content can always be determined, even if the message comes from an intermediate process. It is generally agreed that electronic signatures can be used to achieve these goals (although there are some pitfalls [26]). With electronic signatures, each process p uses its private *signature* σ_p to sign some piece of information v, thereby generating the corresponding *signed information*[9] $\sigma_p(v)$. Without knowing σ_p, it is computationally infeasible (ideally impossible) to compute $\sigma_p(v)$ from v. Recovering the content v of some valid signed information $\sigma_p(v)$ is easy, however, by applying the (usually public) *inverse signature* σ_p^{-1}.

We will place the following specific assumptions on the signature scheme:

(SA1) Only process p can be the originator of $\sigma_p(v)$.

(SA2) Only process p can change the content v of $\sigma_p(v)$.

[9] We will call the process that generated some signed information its *originator*. A *signed message* denotes a message that contains some signed information. Note that the sender of a signed message may be different from the originator of the signed information it contains.

(SA3) The content of any signed information can be extracted at any process in the system. More specifically, we assume that any process can compute $\sigma_p^{-1}(SI) = v$ if and only if $SI = \sigma_p(v)$, whereas $\sigma_p^{-1}(SI) =$ ERROR for some distinguished value ERROR otherwise.

Note that (SA1) implicitly requires countermeasures against replay attacks, since an eavesdropping process could otherwise inject pre-recorded messages from an earlier execution run. This could be avoided by incorporating unique execution sequence numbers in messages, for example. (SA2) implies that forwarding processes cannot manipulate signed messages without being detected at the receiver. (SA3) secures that authentication does not introduce new errors (interpreting a valid signature for an invalid one) or mask present ones (generating an apparently valid v out of an incorrect SI), cf. [26].

Clearly, the above requirements hold only for *unbroken* signatures. If process p's signature has been disclosed (deliberately or not), (SA1) and (SA2) do not hold anymore. More specifically, some other process $q \neq p$ could then manufacture signed messages $\sigma_p(v)$ as if they originated from process p.

In the subsequent sections, we will investigate authenticated Byzantine agreement algorithms that disseminate the transmitter t's value v by forwarding signed messages via paths of distinct processors, one hop per round, as does the non-authenticated algorithm OMH of Section 5.3.1: In round k, a value v that is sent from process $p_1 = t$ along the chain of distinct processes $p_2 \dots p_k$ usually arrives at some receiver q as a multiply signed message $M = \sigma_{p_k} \cdots \sigma_{p_1}(v)$. Due to an omission/value failure, however, it might happen that process $p_{\ell+1} \in \{p_2, \dots, p_k, p_{k+1}\}$ with $p_{k+1} = q$ does not receive a valid round ℓ message $\sigma_{p_\ell} \cdots \sigma_{p_1}(v)$ but rather $\emptyset$ (we assume that manifest faults of messages, like invalid signatures, reception of multiple messages in the same round etc. are mapped to $\emptyset$ as usual). The algorithm ZAr (see Section 5.5.3) simply stops forwarding in this case, so that no message M will arrive at q in this case. Other algorithms like OMHA (Section 5.5.1) and ZA (Section 5.5.2) cause $p_{\ell+1}$ to generate a message $\sigma_{p_{\ell+1}}(R_\emptyset)$ containing a specific value $R_\emptyset$ meaning "I am reporting $\emptyset$" in this case, which is forwarded along the remaining path as usual. Consequently, the final receiver q could get a signed message $\sigma_{p_k} \cdots \sigma_{p_{\ell+1}}(R_\emptyset)$ instead of $M = \sigma_{p_k} \cdots \sigma_{p_1}(v)$ here as well.

All our algorithms explicitly need the string of forwarding process id's $p_k, \dots, p_1$ of any round k message, both for assigning an arriving message to the appropriate concurrent instance of the algorithm (recall Remark 3 on Definition 8) and for applying the appropriate inverse signature when successively recovering the transmitter's value v. Consequently, in a real implementation, $p_k, \dots, p_1$ must somehow be incorporated in signed messages. Moreover, any process $p_{\ell+1}$ generating a message containing $R_\emptyset$ must encode $p_\ell, \dots, p_1$ in $R_\emptyset = R_\emptyset^{p_\ell,\dots,p_1}$. We usually prefer the shorthand notation $R_\emptyset$, however, to keep the notation simple.

The following Definition 9 summarizes the properties of *valid* forwarded messages. A message that is not valid is called *manifest faulty*.

Definition 9 (Valid Messages) *For $k \geq 1$, a message M arriving at some process p via a forwarding path of length k is* valid *if and only if it satisfies the following properties:*

(0) $M = \sigma_{p_k} \cdots \sigma_{p_{\ell+1}}(v)$ for some ℓ with $k > \ell \geq 0$, where all p_i, $k \geq i \geq \ell + 1$, are different and all signatures in M are valid.

(1) If M arrived at p via the link from process q, then $p_k = q$.

(2) If $\ell = 0$, i.e., $M = \sigma_{p_k} \cdots \sigma_{p_1}(v)$, then $p_1 = t$ and v is a legitimate transmitter value.

(3) If $\ell > 0$, then the transmitter has not signed the message and $v = R_\emptyset^{p_\ell,\ldots,p_1}$ (only for some forwarding schemes).

Although faulty processes and links can of course generate non-valid messages, their capabilities in doing so are considerably limited due to authentication. To prove this fact, we first summarize the extensions of our system/failure model due to to authentication:

- Any obedient process adheres to the particular algorithm at all times. Note that manifest faulty messages will usually be discarded at some point[10] and $\emptyset$ be used instead (recall hybrid majority in Section 5.3.1, for example).

- A *symmetric* faulty process may in principle perform arbitrarily internally and may sign and broadcast even a faulty message to all receivers consistently. However, it may neither disclose its signature nor sign and forward (or disseminate otherwise) multiple messages that arrive in the same round.

- An *arbitrary* faulty process may perform arbitrarily. In particular, it may forward multiple messages in the same round, collude with other arbitrary faulty processes, disclose its signature, etc.

- Links are incapable of generating valid signatures. However, an arbitrary faulty link could inject signed messages generated elsewhere in the system.

As an immediate corollary, Definition 9 in conjunction with (SA1)–(SA3) implies that neither faulty forwarding processes nor faulty links can introduce any v' (except $v' \in \{\emptyset, R_\emptyset\}$) different from the value v sent by a not arbitrary faulty transmitter p_1 with unbroken signature: Any valid signed message conveying such a v' must carry p_1's signature, which cannot happen if v' has not been sent by p_1.

Authentication thus allows us to set $f_\ell^{sa} = f_\ell^{ra} = 0$ in our system model, that is, arbitrary link failures need only be counted as omissive ones anymore. In addition, the value X_p actually sent by a symmetric faulty process p should be a valid message (typically with incorrect content v), otherwise p would actually exhibit a manifest fault only. The same is true for arbitrary faulty processes, which would otherwise appear as omission faulty only.

[10] This does not necessarily occur during forwarding, since checking all signatures is a time-consuming task.

In the remainder of this section, we will establish some general results that greatly simplify the analysis of the authenticated algorithms in the subsequent section. The first Lemma 10 shows that any information forwarded via paths with a common prefix that includes at least one not arbitrary faulty process with an unbroken signature is unique.

Lemma 10 (Uniqueness) *Consider any Byzantine agreement algorithm that forwards the transmitter t's value along paths of different processes, one hop per round. Let $M = \sigma_{p_k} \dots \sigma_{p_1}(v)$ resp. $M' = \sigma_{p'_{k'}} \dots \sigma_{p'_1}(v')$ be two valid messages containing v resp. v', which arrive at two obedient processes p resp. p' (possibly $p = p'$) along forwarding paths of length $k \geq c$ resp. $k' \geq c$ with a common prefix of length $c \geq 1$, i.e., $p_1 = p'_1 = t, p_2 = p'_2, \dots, p_c = p'_c$. If at least one of those c processes is not arbitrary faulty and has an unbroken signature, then $v = v'$.*

Proof: Assuming the contrary, some obedient or symmetric faulty process p_x, $1 \leq x \leq c$, must have forwarded two different signed messages, namely, $M_x = \sigma_{p_x} \dots \sigma_{p_1}(v)$ and $M'_x = \sigma_{p'_x} \dots \sigma_{p'_1}(v')$, to its successor(s) on each path in the same round. Since both obedient and symmetric faulty processes may broadcast at most a single message per round according to Definition 2, p_x cannot have generated and sent both M_x and M'_x but at most one of those, say, M_x. Still, M'_x could have been generated out of thin air by an arbitrary link failure hitting the link from $p_x = p'_x$ to p'_{x+1} (with $p'_{k+1} = p'$). This is impossible according to (SA1), however, since M'_x can only be generated by process p_x as it involves σ_{p_x}. $\square$

Of course, Lemma 10 does not rule out the possibility of receiving inconsistent information: Apart from different values disseminated by a Byzantine transmitter, processes p and/or p' might get $\emptyset$'s due to omissions or valid messages containing $R_\emptyset$'s (in case of algorithms like OMHA and ZA). Our lemma does not apply here, since M and M' must have a common prefix, which is impossible if either one (but not both) contains $R_\emptyset$. What is ensured by Lemma 10, however, is that the information routed via any two paths with a common prefix incorporating a single not arbitrary faulty process with an unbroken signature is the same. Consequently, if the common prefix of the forwarding path is long enough, i.e., $k \geq f_a + 1$, all receivers that obtain a valid non-$R_\emptyset$ message agree on the transmitter's value. We can prove an even stronger result, however:

Lemma 11 (Identity) *Consider a Byzantine agreement algorithm with $n > f_\ell^s + f_\ell^r + f_a + f_s + f_o + f_m + 1$ processes, $f_\ell^s, f_\ell^r, f_a, f_s, f_o, f_m \geq 0$, which forwards the transmitter's value along all possible paths of different processes. If an obedient process p receives some v in a valid message $M = \sigma_{p_k} \dots \sigma_{p_1}(v)$, where at least one of the processes in the set $\{p_1, \dots, p_{k-\min\{1, f_\ell^s\}}\}$ is consistent with unbroken signature, then every other non-faulty process q also gets a valid message with at most k signatures that contains v.*

Proof: Assume that p_x is consistent, with $1 \leq x \leq k - \min\{1, f_\ell^s\}$ being the smallest index of consistent nodes, and thus sends $M_x = \sigma_{p_x} \dots \sigma_{p_1}(v)$. Note that p_x cannot be manifest faulty here, since M would not carry σ_{p_x} in this case. Since p_x must have received v from itself in this case, we can restrict our attention to obedient processes q which are not on the path $p_1, \ldots, p_x$. If there are no link failures, i.e., $f_\ell^s = f_\ell^r = 0$, all those obedient processes get $M' = M_x$ and we are done. If $f_\ell^s > 0$, at least $n' \geq n - 1 - f_a - f_s - f_o - f_m - f_\ell^s > f_\ell^r + 1$ non-faulty processes get M_x and forward it. Every obedient process q thus receives at least one of those forwarded messages, despite of the at most f_ℓ^r receive link failures it might experience. □

It is important to note that Lemma 11 cannot be extended to obedient instead of non-faulty processes q, i.e., our lemma does not ensure uniform agreement on non-$\emptyset$ values. More specifically, it could be extended to (strong) manifest processes but not to omission faulty ones: If there is some omission faulty process p_o in M's path prefix $p_1, \ldots, p_{x-1}$, M_x and hence v is not forwarded to p_o by p_x. Unlike in case of a consistent p_o, it cannot be guaranteed that the value contained in M_x has already been self-delivered to p_o either, since $\emptyset$ might have been received instead. Hence, p_o could fail to get some value v present at other obedient nodes.

It only remains to address the question of how to model broken signatures. More specifically, since arbitrary faulty nodes might already know all their signatures, we now assume that the signatures of at most f_b not arbitrary faulty processes are also "common knowledge".

Knowing a signature allows some malicious process to generate a signed message on behalf of any of the f_b processes, say, process p, even during forwarding. An attacker, sitting either on p or on an arbitrary faulty forwarding process q, could hence make p appear arbitrary faulty. However, Lemma 10 and 11 revealed that it cannot do more. Consequently, the problem of incorporating f_b additional broken signatures can be solved by simply increasing f_a to $f_a + f_b$.

Before we can start with the analysis of our authenticated algorithms, we should explain how we integrate the authentication protocols of Chapter 4 into our Byzantine agreement algorithms. It is immediately apparent that we must do so, since the signatures are an ideal denial-of-service attack opportunity. Even if we only check signatures in the last round of the algorithm, an intruder[11] may still inject an arbitrary number of incorrectly signed messages: Processes cannot filter out any messages without verifying the signatures lest they discard a valid message of the algorithm and introduce additional faults. Hence, we must employ our denial-of-service defenses described in Chapter 4. We therefore propose to precede every execution of a Byzantine agreement algorithm with an initialization phase in which each process sets up an individual MAC key with every other process using the authentication protocol of Section 4.2. During the execution of the Byzantine agreement algorithm, a sender process p signs its message as usual, but adds the MAC for receiver q when transmitting the message to q. Process q verifies the MAC on-the-fly during reception and ascertains that the message

[11] Note that we only consider processes which are not part of the process set participating in the algorithm. An intruder participating in the algorithm can of course do everything an arbitrary faulty process is capable of doing.

has indeed been sent by p. By comparing the MACs of new messages to those already received, q can also easily filter out replayed messages from p (although this takes up some memory and computation time). Messages from previous executions, finally, are filtered out by checking the unique sequence number indicating the corresponding execution run. Hence, any message the intruder injects on the link from p to q is filtered out by q at once.

5.5 Hybrid Written Messages Algorithms

Since the power of faulty nodes is considerably restricted when using written messages, authenticated Byzantine agreement algorithms should have much better fault-tolerance capabilities than non-authenticated algorithms. For example, it follows immediately from Lemma 11 that all non-faulty processes get the same set of values $\notin \{\emptyset, R_\emptyset\}$ if the transmitter's value is forwarded via all paths of length $k > f_a + f_o + \min\{1, f_\ell^s\}$. This reveals, for example, that the simple authenticated algorithm SMH of [34], which just forwards the transmitter's value for sufficiently many rounds and then takes the hybrid-majority of all received values, achieves agreement and validity with as few as $n > f_\ell^s + f_\ell^r + f_a + f_s + f_o + f_m + 1$ processes.

In the following subsections, we will provide a detailed analysis of a few alternative Hybrid Written Messages Algorithms.

5.5.1 Algorithm OMHA

In this section, we will analyze a variant of the algorithm OMHA developed in [26] under the system model of Sections 5.1 and 5.4. The original algorithm of [26] is the same as OMH except that every message sent in OMHA(m) with $m > 0$ must be signed. In our variant, messages must be signed in OMHA(0) as well. Moreover, we utilize the signature as the wrapper function R and report missing and manifest faulty messages by a distinguished value $R_\emptyset \neq \emptyset$. Note that $R_\emptyset$ is also the default value returned by hybrid-majority (see Section 5.3.1) if no majority exists.

Definition 10 (Algorithm OMHA [26]) *The Authenticated Hybrid Oral Message algorithm OMHA is defined recursively as follows:*

OMHA(0):

1. *The transmitter t sends its signed value $w_t = \sigma_t(v)$ to every receiver and delivers $v_t = \sigma_t^{-1}(w_t) = v$, or $v_t = \emptyset$ if no or a manifest faulty w_t results from self-reception, recall Remark 2 on OMH's Definition 8.*

2. *Every receiver p delivers $v_p = \sigma_t^{-1}(w_p)$ if w_p was received from the transmitter, or $v_p = \emptyset$ if no or a manifest faulty message was received.*

OMHA(m), $m > 0$:

1. *The transmitter t sends its signed value $w_t = \sigma_t(v)$ to every receiver and delivers $v_t = \sigma_t^{-1}(w_t) = v$, or $v_t = \emptyset$ in case of no or a manifest faulty w_t.*

2. *For every receiver p, let w_p be the value (= signed message) it obtains from the transmitter, or $R_\emptyset$ if no or a manifest faulty message was received.*

   Every receiver p acts as the transmitter in algorithm OMHA$(m-1)$ to communicate w_p to all receivers.

3. *For every p and $q \neq p$, let w_p^q be the value receiver p delivers as the result of OMHA$(m-1)$ initiated by receiver q in step 2 above. Every receiver p calculates the hybrid-majority H_t of all values w_p^q and its own value $w_p^p = w_p$. If $H_t \neq R_\emptyset$, then $\sigma_t^{-1}(H_t)$ is delivered as v_p, else $\emptyset$ is delivered.*

As shown in the previous section, neither faulty processes nor links can generate new values, so the only values that may occur during the execution of OMHA are those originally sent by the transmitter, $\emptyset$, and $R_\emptyset$. Unfortunately, the existence of inconsistent values from arbitrary faulty transmitters in conjunction with $R_\emptyset$ values caused by arbitrary/omission faulty forwarding processes and links is enough to make the performance of OMHA no better than that of OMH — except that link failure tolerance is slightly improved.

Lemma 12 (Validity OMHA) *For any $m \geq min\{1, f_\ell^s\}$ and any f_a, f_s, f_o, f_m, f_ℓ^s, $f_\ell^r \geq 0$, our algorithm OMHA(m) satisfies the validity property (B2) if there are strictly more than $2f_\ell^s + f_\ell^r + 2(f_a + f_s) + f_o + f_m + m$ participating processes.*

Theorem 10 (Agreement OMHA) *For any $m \geq f_a + f_o + min\{1, f_\ell^s\}$ and any f_a, f_s, f_o, f_m, f_ℓ^s, $f_\ell^r \geq 0$, our algorithm OMHA(m) satisfies agreement (B1) and validity (B2) if there are strictly more than $2f_\ell^s + f_\ell^r + 2(f_a + f_s) + f_o + f_m + m$ participating processes.*

Proof: We showed in Section 5.4 that adding authentication amounts to restricting the behavior of faulty processes and links. Since OMHA can be mapped one-to-one to OMH with restricted failures, we can immediately re-use Lemma 8 and Theorem 8 by plugging in modified parameter values. However, whereas ruling out arbitrary link failures yields some improvement over the non-authenticated case, there is no real gain with respect to process-failures: Arbitrary faulty transmitters may still send inconsistent information, and arbitrary faulty forwarding processes may inconsistently send (faulty) information and $R_\emptyset$. Hence, we cannot improve the numbers f_a, f_s etc. of process failures in the authenticated case.

Consequently, we only have to plug in $f_\ell^{sa} = f_\ell^{ra} = 0$ in Lemma 8 and Theorem 8 to obtain our results. □

Remarks:

1. Note that OMHA does not depend critically upon authentication. In fact, it just degrades to OMH even if all signatures are broken, provided we spend f_ℓ^{ra} additional nodes.

2. We already mentioned that the original OMHA in [26] did not sign messages sent in OMHA(0). Recall that (A2) in Definition 2 assumes a point-to-point network where the transmitter of a message can be uniquely identified. If a link failure could only cause an omission or a manifest failure, Theorem 10 would remain valid for the original algorithm as well. However, if a link failure can substitute an $R_\emptyset$ value for the real message, then the original algorithm performs no better than OMH. Hence, by Theorem 8, we would need strictly more than $2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + f_o + f_m + m$ processes for the original version of OMHA.

5.5.2 Algorithm ZA

In this section, we will analyze the authenticated algorithm ZA of [26] under the failure model of Definition 2. ZA has been derived from the flawed algorithm Z of [69] and provides a much better resilience than OMHA. However, its correctness depends critically upon unbroken signatures of not arbitrary faulty nodes (but recall the end of Section 5.4 for how to incorporate this).

Definition 11 (Algorithm ZA [26]) *Let $val(M)$ be a function that extracts the value v contained in a valid signed message $M = \sigma_{p_k} \dots \sigma_{p_1}(v)$, $k \geq 0$[12], by removing all signatures, or returns $val(M) = \emptyset$ if M is manifest faulty or $M = \emptyset$. The algorithm ZA is defined recursively as follows:*

ZA(0):

1. *The transmitter t signs its value v and sends the resulting $w_t = \sigma_t(v)$ to every receiver. The transmitter delivers $v_t = val(w_t) = v$, or $v_t = \emptyset$ if no or a manifest faulty w_t was (self-)received, recall Remark 2 on OMH's Definition 8.*
2. *Every receiver p delivers $v_p = val(w_p)$, where w_p is the signed message received from the transmitter.*

ZA(m), $m > 0$**:**

1. *The transmitter t sends its signed value $w_t = \sigma_t(v)$ to every receiver and delivers $v_t = val(w_t)$.*
2. *For every process p, let w_p be the value p has obtained from the transmitter, or $\emptyset$ if no or a manifest faulty message has been received. Every receiver p acts as the transmitter in algorithm ZA($m-1$) to send w_p to all receivers.*
3. *For every process p and $q \neq p$, let w_p^q be the (unsigned) value process p obtained from receiver q in step (2) using algorithm ZA($m-1$). Every receiver p delivers the hybrid-majority of all w_p^q and its own unsigned value $w_p^p = val(w_p)$; if no majority exists, $\emptyset$ is delivered.*

[12]For $k = 0$, this means $val(v) = v$.

Unlike OMHA, which uses a distinguished reporting value $R_\emptyset$, ZA just uses $R_\emptyset = \emptyset$. Note that $\emptyset$ is also the default value returned by hybrid-majority in the absence of a majority. This implies that the only values that may occur during the execution of ZA are the value(s) sent by the transmitter and $\emptyset$. Consequently, there is no need for an "overwhelming" number of non-faulty values in order to compensate $R_\emptyset$ values. In fact, even a single non-$\emptyset$-value is sufficient here. The following Lemma 13 and Theorem 11 show that ZA satisfies validity (B2) and agreement (B1) with optimal resilience.

Lemma 13 (Validity ZA) *For any $m \geq min\{1, f_\ell^s\}$ and any f_a, f_s, f_o, f_m, f_ℓ^s, $f_\ell^r \geq 0$, algorithm ZA(m) satisfies the validity property (B2) if there are strictly more than $f_\ell^s + f_\ell^r + f_a + f_s + f_o + f_m + 1$ participating processes.*

Proof: Using induction, we show that every non-faulty process delivers the value sent by a consistent transmitter, and/or $\emptyset$-values in case of a manifest/omission faulty transmitter. Recall that considering those cases is sufficient for validity (B2). Lemma 10 ensures that no other value can be delivered; for brevity, we will not mention this explicitly in every step of our proof.

Induction starts at $m = 0$ in the absence of link failures, i.e., $f_\ell^s = f_\ell^r = 0$. In case of a consistent transmitter t sending ν, every non-faulty receiver will obtain $\sigma_t(\nu)$ in ZA(0) and will hence deliver ν (and so does the transmitter); a manifest faulty transmitter causes all receivers and itself to deliver $\emptyset$. In case of an omission faulty transmitter that sends either ν or causes $\emptyset$ to be delivered, every process will deliver either ν or $\emptyset$ in ZA(0) as required.

In case of $f_\ell^s > 0$, induction starts at $m = 1$. In ZA(1), a non-faulty or symmetric faulty transmitter signs and sends its value ν to all $n-1$ receivers, $n' \geq n-1-f_a-f_s-f_o-f_m \geq f_\ell^s+f_\ell^r+1$ of which are non-faulty. In addition, the (non-faulty) transmitter delivers ν. At least $n' - f_\ell^s$ of the receivers will hence receive and broadcast ν in step 1 of ZA(0). Therefore, by the end of ZA(0), every non-faulty receiver gets ν from at least $n' - f_\ell^s - f_\ell^r \geq 1$ processes and will hence deliver it. Recall that any transmitter also uses its own value in step 3 of ZA(0), cf. Remark 2 on Definition 8. As soon as a process has obtained at least one ν as the result of ZA(0), however, it will deliver it as the result of ZA(1) as well.

If the transmitter in ZA(1) is manifest faulty, no receiver can get a valid w_p since such a message must contain t's signature. Therefore, only $\emptyset$-values can be forwarded and delivered in ZA(0), and ZA(1) must return $\emptyset$ here as required. Finally, if the transmitter is omission faulty, any non-faulty process can only deliver ν or $\emptyset$ since these are the only possible values. Validity is hence fulfilled in this case as well.

For the induction step, we assume that our lemma holds for ZA$(m-1)$ with $m-1 \geq$ min$\{1, f_\ell^s\}$ and show that it also holds for ZA(m). Using the same argument as above, a non-faulty or symmetric faulty transmitter signs and sends its value ν to all $n-1$ receivers, $n' \geq n-1-f_a-f_s-f_o-f_m \geq f_\ell^s+f_\ell^r+1$ of which are non-faulty. In addition, a (non-faulty) transmitter also delivers ν. In case of a manifest faulty transmitter, no receiver can get a valid w_p; so any receiver could forward only $\nu = \emptyset$. By the induction hypothesis, all non-faulty receivers will therefore deliver ν as the result of ZA$(m-1)$

and hence, due to hybrid-majority in conjunction with Lemma 10, as the result of ZA(m) as well. If the transmitter is omission faulty, any non-faulty process can only deliver ν or $\emptyset$ since these are the only valid values, so validity is fulfilled in this case as well. □

Theorem 11 (Agreement and Validity ZA) *For any $m \geq f_a + f_o + min\{1, f_\ell^s\}$ and any f_a, f_s, f_o, f_m, f_ℓ^s, $f_\ell^r \geq 0$, algorithm ZA(m) satisfies agreement (B1) and validity (B2) if there are strictly more than $f_\ell^s + f_\ell^r + f_a + f_s + f_o + f_m + 1$ participating processes.*

Proof: As argued in Remark 1 on Theorem 8, the proof is the same as that for agreement in OMH. □

Remarks:

1. We slightly modified the original definition of ZA in [26], which lacked a few details. First, every transmitter in recursive instances must send its value to itself and use it as additional input to hybrid-majority; it does of course not further participate in the recursion. Secondly, our algorithm removes the whole signature chain in ZA(0) at once before delivery.

2. Unlike OMHA, which requires signatures also in the final instance OMHA(0) to cope with link failures in the last round, ZA would work without it. In fact, in case of $f_\ell^s > 0$, any non-$\emptyset$ signed message submitted to ZA(0) in ZA's execution bears already $m \geq f_a + f_o + 1$ signatures, i.e., at least one from a consistent process. By Lemma 10, this is sufficient to ensure agreement — adding another signature within ZA(0) is hence not required.

3. Although ZA is defined recursively like OMHA, its execution develops quite differently: According to Lemma 11, every process gets sufficient information to achieve validity within the first two rounds of ZA(m) in case of a not arbitrary faulty transmitter, regardless of the number of rounds actually employed. Validity of OMHA(m), however, is achieved only after its full number of $m+1$ rounds. Moreover, m needs to be included into n for OMHA, since faulty transmitters can inject $R_\emptyset$'s in additional rounds that must be balanced. This is not true for ZA, since the latter faces valid signed messages containing ν and $\emptyset$ only.

5.5.3 Algorithm ZAr

Even though ZA has a much better resilience than OMHA, it does not fully exploit the benefits of authenticated messages. More specifically, we know from Lemma 11 that every non-faulty receiver obtains the same set of distinct (unsigned) values $\neq \emptyset$ at the end of ZA(0). There is no need, however, to receive (and hence forward) multiple signed messages containing a specific value v. Moreover, there is no use in forwarding $\emptyset$-values, since they are discarded at the end anyway. Note that item (3) in Definition 9 is hence void here.

The algorithm ZAr given in Definition 12 below is based upon those observations. It dramatically reduces both message and computational complexity of ZA, from exponential to polynomial.

Definition 12 (Algorithm ZAr) *Let $\mathcal{W}_p$, initially $\mathcal{W}_p = \{\}$, be the set of legitimate unsigned values already seen by process p during the execution. Moreover, let $val(M)$ be the value v contained in a valid signed message after removing all signatures, and $val(M) = \emptyset$ otherwise.*

ZAr(0):

1. *The transmitter t sends the signed message $w_t = \sigma_t(v)$ to every receiver, and adds $val(w_t)$ to $\mathcal{W}_t$ if $val(w_t) \neq \emptyset$.*
2. *For every receiver p, if p has received a valid message w_p containing a value $\emptyset \neq val(w_p) \notin \mathcal{W}_p$, then p adds $val(w_p)$ to $\mathcal{W}_p$.*

ZAr(k), $m \geq k > 0$**:**

1. *The transmitter t sends the signed message $w_t = \sigma_t(v)$ to every receiver, and adds $val(w_t)$ to $\mathcal{W}_t$ if $val(w_t) \neq \emptyset$.*
2. *For every receiver p, if p has received a valid message w_p that satisfies $\emptyset \neq val(w_p) \notin \mathcal{W}_p$, then p acts as the transmitter in algorithm ZAr($k-1$) to disseminate w_p to all receivers.*
3. *End of ZAr(m) only: For every process q, if $\mathcal{W}_q$ contains a single legitimate value v_q, then q delivers v_q, otherwise it delivers $\emptyset$.*

Using Lemma 11, it is not difficult to show that ZAr satisfies validity and agreement:

Lemma 14 (Validity ZAr) *For any $m \geq min\{1, f_\ell^s\}$ and any f_a, f_s, f_o, f_m, f_ℓ^s, $f_\ell^r \geq 0$, algorithm ZAr(m) satisfies validity (B2) if there are strictly more than $f_\ell^s + f_\ell^r + f_a + f_s + f_o + f_m + 1$ participating processes.*

Proof: By Lemma 11, we know that $min\{1, f_\ell^s\}+1 \leq 2$ rounds suffice to ensure that every obedient process gets the transmitter's value v if the transmitter is non-faulty or symmetric faulty. If the transmitter is manifest faulty, no receiver ever gets a non-$\emptyset$ value. In case of an omission faulty transmitter, Lemma 10 ensures that $\emptyset$ and v are the only possible values. Hence, validity is always fulfilled. □

Theorem 12 (Validity and Agreement ZAr) *For any $m \geq f_a + f_o + min\{1, f_\ell^s\}$ and any f_a, f_s, f_o, f_m, f_ℓ^s, $f_\ell^r \geq 0$, algorithm ZAr(m) satisfies validity (B2) and agreement (B1) if there are strictly more than $f_\ell^s + f_\ell^r + f_a + f_s + f_o + f_m + 1$ participating processes.*

Proof: Due to Lemma 14, it only remains to show agreement. In our proof, we use the fact that every non-faulty process obtains the same set of values at the end of round $m + 1$ by Lemma 11. Note carefully that we only have to look at values that "survive" forwarding until ZAr(0), because forwarding extincts only if a particular value is already contained in some process's $\mathcal{W}_p$ — but in that case, the earlier message must have already been forwarded. Note carefully that it does not matter here that any two signed message containing the same v are forwarded along different paths: Although the set of receivers are different, they differ only in the nodes on the already taken paths. All non-faulty ones among those, however, must have got v by self-reception and, consequently, do not need forwarding.

Now, the signature chains in the final round have length $m + 1 \geq f_a + f_o + 1 + min\{1, f_\ell^s\}$. Consequently, there must be at least one consistent process p_x in each such chain $p_1, \ldots, p_{m+1}$ with $1 \leq x \leq m + 1 - \min\{1, f_\ell^s\}$. If p_x is non-faulty or symmetric faulty, Lemma 11 guarantees agreement. If p_x is manifest faulty, no signed message with $m+1$ signatures incorporating σ_{p_x} can be received at any process. Hence, every non-faulty process obtains the same set of non-$\emptyset$ values. □

Remarks:

1. During the first two rounds, ZAr sends the same number of non-$\emptyset$ messages as ZA. In subsequent rounds, however, ZAr is clearly superior with respect to communication complexity: From Definition 12, it is obvious that every value v sent by the transmitter is forwarded by any node at most once to all $n-1$ receivers. Hence, every v causes at most $(n - 1)^2$ message during forwarding, so the worst case occurs if the transmitter sends each of its $n - 1$ receivers a different value; recall that no other value except $\emptyset$ can be generated in the system. Consequently, at most $n - 1 + (n - 1)^3$ messages can be sent system-wide.

2. If the initial transmitter is arbitrary faulty and sends multiple values, then ZAr will always deliver $\emptyset$. However, ZA might deliver one of the sent values $v \neq \emptyset$ in this case, if v has been received by a majority of processes.

3. The reduced algorithm ZAr is in fact the same as the algorithm for atomic broadcast under Byzantine failures proposed in [11]. The latter was analyzed in a more abstract failure model, however, where the only requirement is that the removal of faulty processes and links does not partition the network. For example, non-faulty processes might even be connected in a chain there, in which case the atomic broadcast algorithm would need $n - 1$ rounds. By contrast, we showed in Theorem 2 that our link failure model ensures that any two non-faulty processes are connected to each other by at least one non-faulty path of length $\min\{1, f_\ell^s\} + 1 \leq 2$, which explains the comparatively small number of rounds required by ZAr.

5.6 Broadcast Networks

The consensus algorithms analyzed in this thesis assume a point-to-point network, which implies that the sender of a message is always known. If we use those algorithms on a broadcast network, however, the sender is not necessarily known: If we do not sign messages, we obviously lose the ability to identify the sender of a message, thus allowing faulty processes to impersonate non-faulty ones. Hence the oral messages algorithms would not work in this case. Written messages algorithms, however, should reasonably[13] work because they do not allow impersonation. To be precise, any oral messages algorithm that achieves consensus under the system model of Definition 2 will achieve consensus in a broadcast network if authentication is added. This is due to the fact that using a broadcast network only violates assumption (A2), i.e., the sender is not known. With authentication, (A2) is ensured again and the system model remains the same. In fact, without link failures, written messages algorithms would even benefit from the broadcast network, because neither arbitrary nor omission faulty processes are possible anymore. Since every process sends only one message, which is automatically broadcast to all processes, every receiver must get the same value. So we can in fact set $f_a = f_o = 0$ and count all arbitrary failures as symmetric failures and all omission failures as manifest failures for any written messages algorithm analyzed under the hybrid failure model.

If link failures are possible, however, we find that they now have a lot more power than before. Whereas they can simply be caught by adding an appropriate multiple of f_ℓ^r and f_ℓ^s to the number of processes in the point-to-point case, we experience the unpleasant effect that they make arbitrary (but not omission) failures possible in the broadcast case [53]. However, the behavior of arbitrary processes is restricted:

Consider a message from an arbitrary faulty process which is not received by f_ℓ^s receivers. If that process sends a second message containing a different value, which is not received by another f_ℓ^s receivers, then at most $2f_\ell^s$ receivers will only get one message and will assume that the message is valid. The other processes do detect the second message from the same sender and will use the value $\emptyset$ due to the manifest failure. So the obvious solution is either to count arbitrary failures again, or to count sender link failures twice, i.e., require $4f_\ell^s$ instead of $2f_\ell^s$ additional processes.

When analyzing OMHA in broadcast networks, we can exploit the restricted behavior of arbitrary processes. First, it is easily seen that the validity proof can be taken over unchanged: Since the transmitter is not arbitrary faulty, and since arbitrary faulty processes do not occur in the last but one line of equations (5.3) and (5.4) in the proofs of Lemma 7 and Lemma 8, their different behavior in the broadcast network has no impact on validity and the proof of Lemma 8 still holds. As far as omission faulty processes are concerned, they either behave non-faulty or like manifest failures. In any case, all obedient receivers will deliver the same value for them.

[13] Apart from the problem of jamming.

Theorem 13 (Agreement and Validity) *For any f_a, f_s, f_o, f_m, f_ℓ^s, $f_\ell^r \geq 0$ and any $m \geq \min\{1, f_\ell^s\}$, OMHA(m) satisfies agreement (B1) and validity (B2) if there are strictly more than $4f_\ell^s + f_\ell^r + 2(f_a + f_s) + f_o + f_m + m$ participating processes.*

Proof: Again, we only look at the case where the transmitter behaves arbitrary faulty as described above.

Let $m = 1$. Abbreviating the number of initially participating receivers with $n' \geq 4f_\ell^s + f_\ell^r + 2(f_a + f_s) + f_o + f_m + m$, the transmitter sends both ν and ν', which are received and turned into $\emptyset$ by at least $n' - 2f_\ell^s - (f_a - 1) - f_s - f_o' - f_m'$ non-faulty receivers, whereas at most f_ℓ^s processes receive only ν and f_ℓ^s only ν'. Here, $f_o' \leq f_o$ is the number of omission faulty receivers that will commit a manifest failure in OMHA(0) (all others will appear like non-faulty processes) and $f_m' \leq f_m$ is the actual number of manifest faulty processes.

In OMHA(0), a non-faulty receiver gets $R(\emptyset)$ from at least $n_q' = n' - 2f_\ell^s - (f_a - 1) - f_s - f_o' - f_m' - f_\ell^{r\prime}$ processes, with $f_\ell^{r\prime} \leq f_\ell^r$ link failures according to (A1^r), and it receives at most $n_q'' = n' - f_\ell^{r\prime} - f_o' - f_m'$ values different from $\emptyset$. Therefore, we have

$$\begin{aligned}
2n_q' - n_q'' &= n' - 4f_\ell^s - 2f_a + 2 - 2f_s - f_o' - f_m' - f_\ell^{r\prime} \\
&\geq (f_\ell^r - f_\ell^{r\prime}) + (f_o - f_o') + (f_m - f_m') + m + 2 \\
&> 0
\end{aligned}$$

and $R(\emptyset)$ will win the hybrid-majority on every non-faulty process.

Let us now consider $m > \min\{1, f_\ell^s\}$. At least $\overline{n}_q' = n' - 2f_\ell^s - (f_a - 1) - f_s - f_o' - f_m'$ non-faulty receivers will get $\emptyset$ in OMHA(m) and thus will disseminate $R(\emptyset)$ in OMHA($m-1$). Validity ensures that every obedient process will deliver $R(\emptyset)$ for them. The f_o' omission faulty processes will appear crashed in OMHA($m - 1$) and cause $\emptyset$. Therefore, every obedient process will deliver at most $\overline{n}_q'' = n' - f_o' - f_m'$ values different from $\emptyset$. Since

$$\begin{aligned}
2\overline{n}_q' - \overline{n}_q'' &= n' - 4f_\ell^s - 2f_a - 2f_s - f_o' - f_m' \\
&\geq (f_o - f_o') + (f_m - f_m') + f_\ell^r + m \\
&> 0,
\end{aligned}$$

$R(\emptyset)$ will again win the hybrid-majority on all obedient processes, hence all obedient processes will deliver $\emptyset$. $\square$

The algorithm ZA, however, cannot exploit the different behavior of arbitrary failures so easily. Here, arbitrary faulty processes must still be counted in m. The reason is obvious from the proof of Theorem 13: We have utilized the fact that every non-faulty process receives enough $R(\emptyset)$ values to win the hybrid-majority. This has saved us from using enough rounds to ensure that all processes get exactly the same input set for the hybrid-majority. In ZA, however, only $\emptyset$ does exist, which is not considered in the majority function. Therefore, we will again have to ensure that all processes work with the same input set in ZA(0), effectively requiring $m \geq f_a + \min\{1, f_\ell^s\}$.

Remarks:

1. Note that an arbitrary faulty process can do the worst damage by sending two messages. With a third message, again only f_ℓ^s receivers might not detect a manifest failure.

2. When executing OMHA on a broadcast network, the tradeoff between the point-to-point algorithm of Theorem 10 and the broadcast version of Theorem 13 is $f_a + f_o$ vs. $2f_\ell^s$ additional processes and $f_a + f_o + 2$ vs. 2 rounds.

3. Since ZA is already optimal and treats arbitrary failures like symmetric failures anyway, it will not really benefit from the broadcast network. There is, however, the slight tradeoff between the original version and the broadcast version with $2f_\ell^s$ more processes and f_o less rounds.

Chapter 6

Conclusions and Future Work

6.1 Authentication

We proposed a way to achieve authenticated message exchange in an ad-hoc network while keeping the possibilities for denial-of-service attacks at a minimum. We argued that even though the whole system may be asynchronous, we can nevertheless assume synchronous subsystems which contain enough servers of the certification authority to guarantee synchronous and fault-tolerant communication between nodes and the CA.

We found that substantial hardware support in the form of message filters helps to keep DoS attacks at bay, although it does not completely prohibit DoS attacks in W2F. In order to use our timestamp filter, nodes must know a lot about the system like message transmission delays or the global clock synchronization precision π.

Due to the ad-hoc nature of the network, violations of our system assumptions are possible, and in this case we may even get into situations where authentication cannot be guaranteed anymore. The problem even increases with mobile nodes which may be disconnected from the network for a longer period of time. If violations of our clock synchronization and message delay assumptions can occur, we must decide between security and quality of service: Either we use the timestamp filter, accepting that if the assumptions are unknowingly violated, we may experience network partitions which may even lead to authentication being corrupted if old keys are used, or we do not use the filter and thereby accept a higher impact of DoS attacks.

There are several directions of future work in this area: Since the quality of our algorithms stands and falls with our assumptions, we will continue to search for algorithms which require less assumptions than our current ones while providing us with the same protection against DoS attacks. We will also compute the probability of violating our assumptions and investigate if there are ways to reduce it to zero.

Furthermore, we will subject our final algorithms to a formal analysis to prove their correctness. As experience has shown, even simple algorithms tend to contain subtle flaws which are only revealed through a rigorous formal verification.

Finally, we will evaluate the effectiveness of the denial-of-service defenses employed by our final algorithms using the cost-based framework of Meadows [39]. This framework allows to compute the costs of executing a protocol up to a given point for both the attacker and the defender. It is therefore an adequate means of determining what costs servers and clients bear under attack, and how these costs relate to the costs of the attacker.

6.2 Consensus

We showed that deterministic consensus in presence of link failures is possible, despite the impossibility result of [27]. The latter is avoided by limiting the maximum number of link failures in the broadcast resp. reception of any process. Our perception-based hybrid failure model grants every process at most f_ℓ^r independent receive link failures (with at most f_ℓ^{ra} non-omission failures among those) and f_ℓ^s broadcast link failures in each round, in addition to at most f_a, f_s, f_o, f_m arbitrary, symmetric, omission, and manifest process failures. For $m \geq f_a + f_o + 1$, we analyzed three existing $m+1$-round Byzantine agreement algorithms under this failure model, namely, the non-authenticated OMH as well as its authenticated variants OMHA and ZA. Their respective number of processes was shown to be

$$
\begin{aligned}
n &> 2f_\ell^s + f_\ell^r + f_\ell^{ra} + 2(f_a + f_s) + f_o + f_m + m,\\
n &> 2f_\ell^s + f_\ell^r + 2(f_a + f_s) + f_o + f_m + m,\\
n &> f_\ell^s + f_\ell^r + f_a + f_s + f_m + 1.
\end{aligned}
$$

We also proposed and analyzed a slightly modified OMH which achieves agreement and validity even for obedient processes within one more round.

We provided lower bounds for the required number of processes in presence of omission resp. arbitrary link failures, which are matched by ZA resp. OMH and are therefore tight.

Our results also reveal that the usefulness of authentication depends heavily upon the particular algorithm used. In fact, a consensus algorithm should be specifically designed for using written messages and not simply be adapted from an oral messages solution: Whereas OMHA did not profit much from authentication, ZA benefits considerably— but also depends critically upon its strength. It turned out, however, that both algorithms can withstand intrusions to some extent: In case of broken signatures, OMHA degrades to OMH and hence requires an additional f_ℓ^{ra} in the number of processes. For ZA, a process with a compromised signature must be considered as arbitrary faulty and therefore counted in f_a. As far as link failures are concerned, authentication serves to identify and tolerate link value failures: Any algorithm that requires $f_\ell^r + f_\ell^{ra}$ processes to tolerate link failures will only require f_ℓ^r processes in the authenticated version. Apart from that, authentication is the only means to (more or less) safely employ algorithms like OMHA on top of broadcast networks.

There are two primary directions of current/future research in this area: First, we have already made progress in the analysis of less message-costly consensus algorithms [7] under the perception-based failure model, which even includes algorithms for asynchronous systems. Secondly, there are many applications like intrusion-tolerant admission control, on-line diagnosis, distributed database commitment, etc. that could benefit from the possibility to achieve consensus in systems with link failures. A major part of our future research will hence be devoted to the exploration of such applications.

Bibliography

[1] Hosame Abu-Amara and Jahnavi Lokre. Election in asynchronous complete networks with intermittent link failures. *IEEE Transactions on Computers*, 43(7):778–788, July 1994.

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

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

[4] Tuomas Aura, Pekka Nikander, and Jussipekka Leiwo. DOS-resistant authentication with client puzzles. In *Proceedings of the 8th International Workshop on Security Protocols*, LNCS 2133, pages 170–177, April 2000.

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

[6] P. Barker and AC. Boucouvalas. Performance analysis of the IrDA protocol in wireless communications. In *Proceedings of First International Symposium on Communication Systems and Digital Signal Processing*, volume 1, pages 6–9. IEEE, 1998.

[7] Martin Biely and Ulrich Schmid. Message-efficient consensus in presence of hybrid node and link faults. Technical Report 183/1-116, Department of Automation, Vienna University of Technology, August 2001.

[8] Antoon Bosselaers, René Govaerts, and Joos Vandewalle. Fast hashing on the Pentium. In *Advances in Cryptology, Proceedings Crypto'96*, LNCS 1109, pages 298–312. Springer Verlag, 1996.

[9] CERT. CERT advisory: TCP SYN flooding and IP spoofing attacks. Technical Report CA-1996-21, Carnegie Mellon Software Engineering Institute, September 19, 1996. http://www.cert.org/advisories/CA-1996-21.html.

[10] Bernadette Charron-Bost and André Schiper. Uniform consensus harder than consensus. Technical Report DSC/2000/028, École Polytechnique Fédérale de Lausanne, Switzerland, May 2000.

[11] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Proceedings 15th Int. Conf. on Fault-Tolerant Computing (FTCS-15)*, pages 200–206, Ann Arbor, Michigan, USA, 1985.

[12] Joan Daemen and Vincent Rijmen. AES proposal: Rijndael, September 1999.

[13] T. Dierks and C. Allen. The TLS protocol, version 1.0, January 1999. RFC 2246.

[14] W. Diffie and M. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, 1976.

[15] Hans Dobbertin, Antoon Bosselaers, and Bart Preneel. RIPEMD-160: A strengthened version of RIPEMD. In *Proceedings of Fast Software Encryption (FSE'96)*, LNCS 1039, pages 71–82. Springer Verlag, 1996.

[16] FIPS. Digital Signature Standard (DSS), FIPS PUB 186-2, January 27, 2000.

[17] FIPS. Advanced Encryption Standard, FIPS PUB 197, November 2001.

[18] FIPS. Secure Hash Standard, FIPS PUB 180-2, August 1, 2002.

[19] M. Fischer, N. Lynch, and M. Merrit. Easy impossibility proofs for the distributed consensus problem. *Distributed Computing*, 1(1):26–39, 1986.

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

[21] Jeff Foerster, Evan Green, Srinivasa Somayazulu, and David Leeper. Ultra-wideband technology for short- or medium-range wireless communication. *Intel Technology Journal*, Q2:(11 pages), 2001.

[22] Klaus Gaarder and Einar Snekkenes. Applying a formal analysis technique to the CCITT X.509 strong two-way authentication protocol. *Journal Of Cryptology*, 3:81–98, 1991.

[23] Taher El Gamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In *Advances in Cryptology: Proceedings of CRYPTO 1984*, LNCS 0196, pages 10–18, 1985.

[24] Lee Garber. Denial-of-service attacks rip the Internet. *IEEE Computer*, 33(4):12–17, April 2000.

[25] Li Gong. Efficient network authentication protocols: Lower bounds and optimal implementations. *Distributed Computing*, 9(3):131–145, 1995.

[26] Li Gong, Patrick Lincoln, and John Rushby. Byzantine agreement with authentication: Observations and applications in tolerating hybrid and link faults. In *Proceedings Dependable Computing for Critical Applications-5*, pages 139–157, Champaign, IL, September 1995.

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

[28] Günther Gridling. On the problem of internal interference in CDMA-based ad-hoc networks. Technical Report 183/1-121, Department of Automation, Vienna University of Technology, June 2002.

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

[30] R. Housley, W. Polk, W. Ford, and D. Solo. RFC 3280: Internet X.509 public key infrastructure certificate and certificate revocation list (CRL) profile, April 2002. http://www.ietf.org/rfc/rfc3280.txt.

[31] ISO/IEC. ITU-T Recommendation X.509, ISO/IEC 9594-8: Information technology – open systems interconnection – the directory: Public key and attribute certificate frameworks, May 2001. http://www-t.zhwin.ch/it/ksy/Block08/ITU/X509_4thEdition-DraftV8.pdf.

[32] Ari Juels and John Brainard. Client puzzles: A cryptographic countermeasure against connection depletion attacks. In *Proceedings of the 1999 Network and Distributed System Security Symposium (NDSS'99)*, 1999.

[33] Xuejia Lai, James L. Massey, and Sean Murphy. Markov ciphers and differential cryptanalysis. In *Advances in Cryptology - Eurocrypt'91*, LNCS 547, pages 17–38. Springer-Verlag, 1991.

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

[35] Patrick Lincoln and John Rushby. A formally verified algorithm for interactive consistency under a hybrid fault model. In *Proceedings Fault Tolerant Computing Symposium 23*, pages 402–411, Toulouse, France, June 1993.

[36] Gavin Lowe. Breaking and fixing the Needham-Schroeder public-key protocol using FDR. In Tiziana Margaria and Berhard Steffen, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 2nd International Workshop TACAS'96*, LNCS 1055, pages 147–166. Springer Verlag, March 27–29, 1996.

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

[38] Kanta Matsuura and Hideki Imai. Modification of Internet Key Exchange resistant against denial-of-service. In *Pre-Proc. of Internet Workshop 2000 (IWS2000)*, pages 167–174, February 2000.

[39] Catherine Meadows. A cost-based framework for analysis of denial of service in networks. *Journal of Computer Security*, 9(1-2):143–164, 2001.

[40] Catherine A. Meadows. Analyzing the Needham-Schroeder public key protocol: A comparison of two approaches. In Elisa Bertino, Helmut Kurth, Giancarlo Martella, and Emilio Montolivo, editors, *4rd European Symposium on Research in Computer Security, ESORICS'96*, LNCS 1146, pages 351–364, September 25–27, 1996.

[41] David L. Mills. NTP timescale and leap seconds. http://www.eecis.udel.edu/˜ntp/-ntp_spool/html/leap.htm.

[42] Roger Needham and M. D. Schroeder. Authentication revisited. *ACM Operating Systems Review*, 21(1):7, January 1987.

[43] B. Clifford Neuman and Stuart G. Stubblebine. A note on the use of timestamps as nonces. *ACM Operating Systems Review*, 27(2):10–14, April 1993.

[44] Clifford Neuman, John Kohl, and Theodore T'so. The Kerberos network authentication service (V5), March 10, 2000. INTERNET-DRAFT, Expires September 10, 2000.

[45] Kenneth J. Perry and Sam Toueg. Distributed agreement in the presence of processor and communication faults. *IEEE Transactions on Software Engineering*, SE-12(3):477–482, March 1986.

[46] S. S. Pinter and I. Shinahr. Distributed agreement in the presence of communication and process failures. In *Proceedings of the 14th IEEE Convention of Electrical & Electronics Engineers in Israel*. IEEE, March 1985.

[47] David Powell. Failure mode assumptions and assumption coverage. In *Proc. 22nd IEEE Int. Symp. on Fault-Tolerant Computing (FTCS-22)*, pages 386–395, Boston, MA, USA, 1992. (Revised version available as LAAS-CNRS Research Report 91462, 1995).

[48] Livio Ricciulli, Patrick Lincoln, and Pankaj Kakkar. TCP SYN flooding defense. In *Simulation Multiconference CNDS'99*, January 17–20 1999.

[49] Ronald L. Rivest. RFC 1321: The MD5 message-digest algorithm, April 1992.

[50] Ronald L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(12):120–126, February 1978.

[51] John Rushby. Formal verification of hybrid Byzantine agreement under link faults. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 2001. Available at http://www.csl.sri.com/~rushby/abstracts/byzlinks01.html.

[52] Hasan M. Sayeed, Marwan Abu-Amara, and Hosame Abu-Amara. Optimal asynchronous agreement and leader election algorithm for complete networks with Byzantine faulty links. *Distributed Computing*, 9(3):147–156, 1995.

[53] Ulrich Schmid. Synchronized Universal Time Coordinated for distributed real-time systems. *Control Engineering Practice*, 3(6):877–884, 1995. (Reprint from Proceedings 19th IFAC/IFIP Workshop on Real-Time Programming (WRTP'94), Lake Reichenau/Germany, 1994, p. 101–107.).

[54] Ulrich Schmid. Basic features of the wireline/wireless factory/facility fieldbus. Technical Report 183/1-95, Department of Automation, Vienna University of Technology, August 1999.

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

[56] Ulrich Schmid, Martin Horauer, and Nikolaus Kerö. How to distribute GPS-time over COTS-based LANs. In *Proceedings of the 31th IEEE Precise Time and Time Interval Systems and Application Meeting (PTTI'99)*, pages 545–560, Dana Point, California, December 1999.

[57] Ulrich Schmid and Bettina Weiss. Consensus with oral/written messages: Link faults revisited. Technical Report 183/1-110, Department of Automation, Vienna University of Technology, February 2001.

[58] Ulrich Schmid and Bettina Weiss. Consensus with oral/written messages: Link faults revisited. Technical Report 183/1-124, Department of Automation, Vienna University of Technology, November 2002.

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

[60] Bruce Schneier. *Applied Cryptography: Protocols, Algorithms, and Source Code in C.* John Wiley & Sons, second edition, 1996.

[61] Klaus Schossmaier. *Interval-based Clock State and Rate Synchronization.* Dissertation, Vienna University of Technology, Faculty of Technical and Natural Sciences, 1998. (Appeared in Dissertationen der Technischen Universität Wien, Band 87. Österreichischer Kunst- und Kulturverlag, Wien, December 1997, ISBN 3-85437-201-9.).

[62] Vitaly Shmatikov and John C. Mitchell. Analysis of a fair exchange protocol. In *Proceedings of the 1999 FLoC Workshop on Formal Methods and Security Protocols*, Trento, Italy, 1999.

[63] Gurdip Singh. Leader election in the presence of link failures. *IEEE Transactions on Parallel and Distributed Systems*, 7(3):231–236, March 1996.

[64] Hin-Sing Siu, Yeh-Hao Chin, and Wei-Pang Yang. Byzantine agreement in the presence of mixed faults on processors and links. *IEEE Transactions on Parallel and Distributed Systems*, 9(4):335–345, April 1998.

[65] Sean W. Smith and Steve Weingart. Building a high-performance, programmable secure coprocessor. *Computer Networks*, 31:831–860, 1999.

[66] Dawn X. Song and Adrian Perrig. Advanced and authenticated marking schemes for IP traceback. In *Proceedings IEEE Infocom 2001*, April 22-26, 2001.

[67] T. K. Srikanth and Sam Toueg. Optimal clock synchronization. *Journal of the ACM*, 34(3):626–645, July 1987.

[68] Bernd Thallner and Günther Gridling. Simulation of a wireless CDMA ad hoc network. In *Communications and Computer Networks (CCN'02)*, November 4–6, 2002.

[69] P. M. Thambidurai and Y. K. Park. Interactive consistency with multiple failure modes. In *Proceedings 7th Reliable Distributed Systems Symposium*, October 1988.

[70] Gene Tsudik. Message authentication with one-way hash functions. *ACM SIGCOMM Computer Communication Review*, 22(5):29–38, October 1992.

[71] Chris J. Walter and Neeraj Suri. The customizable fault/error model for dependable distributed systems. *Theoretical Computer Science*, 2000. (Special issue on Dependable Computing, to appear).

[72] Haining Wang, Danlu Zhang, and Kang G. Shin. Detecting SYN flooding attacks. In *IEEE Infocom 2002*, June 23-27, 2002.

[73] Bettina Weiss. Authentication under denial-of-service attacks. In *Communications and Computer Networks (CCN'02)*, November 4–6, 2002.

[74] Bettina Weiss. Authentication under denial-of-service attacks. Technical Report 183/1-122, Department of Automation, Vienna University of Technology, July 2002.

[75] Bettina Weiss and Ulrich Schmid. Consensus with written messages under link faults. In *20th Symposium on Reliable Distributed Systems (SRDS'01)*, pages 194–197, October 28–31, 2001.

[76] Michael J. Wiener. Performance comparison of public-key cryptosystems. *CryptoBytes Technical Newsletter*, 4(1):1–5, Summer 1998. ftp://ftp.rsasecurity.com/pub/cryptobytes/crypto4n1.pdf.

[77] Yongguang Zhang and Wenke Lee. Intrusion detection in wireless ad-hoc networks. In *Proceedings of the 6th Conference on Mobile Computing and Networking*, pages 275–283, August 6-11, 2000.

[78] Lidong Zhou. *Towards Fault-Tolerant and Secure On-Line Services*. PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY USA, May 2001.

[79] Lidong Zhou and Zygmunt J. Haas. Securing ad hoc networks. *IEEE Network*, 13(6):24–30, 1999.

[80] Lidong Zhou, Fred B. Schneider, and Robbert van Renesse. COCA: A secure distributed on-line certification authority. Technical Report TR2000-1828, Computer Science Department, Cornell University, December 2000.

[81] Reto Zimmermann, Andreas Curiger, Heinz Bonnenberg, Hubert Kaeslin, Norbert Felber, and Wolfgang Fichtner. A 177 Mbit/s VLSI implementation of the International Data Encryption Algorithm. *IEEE Journal of Solid-State Circuits*, 29(3):303–307, March 1994.

Curriculum Vitae

Bettina Weiss
Feldgasse 21
A-2542 Kottingbrunn

Personal Data

Date of Birth:	November 15, 1970
Place of Birth:	Mödling, Lower Austria
Nationality:	Austria

Education

1977-1980	Elementary School A-2542 Kottingbrunn, Franz Nagl Gasse 20
1981-1989	High-School (neusprachlicher Zweig) Bundesgymnasium und Bundesrealgymnasium A-2500 Baden, Frauengasse 3-5
1989-1997	Technical University Vienna Computer Science
June 1997	Diploma Thesis *Simulation Environment for Clock Synchronization*

Work Experience

August 1992, July 1993	Ferialpraxis Siemens AG Österreich
Summer Terms 1992-2001	Tutor for the course “Prozeßautomatisierung”
Winter Terms 1993, 2001	Tutor for “Einführung in die Technische Informatik”
1995-September 1999	Research Assistant in FWF/START-Project SynUTC Department of Automation, TU Vienna
September 1999-present	Research Assistant in START-Project W_2F Department of Automation, TU Vienna

Vienna, October 2002