The Dissertation Committee for Amitanand Swaminathan Aiyer certifies that this is the approved version of the following dissertation:

Alternative implementations for storage and communication abstractions in distributed systems

Committee:

Lorenzo Alvisi, Supervisor

Rida A. Bazzi

Michael D. Dahlin

Mohamed G. Gouda

Jay J. Wylie

Alternative implementations for storage and communication abstractions in distributed systems

by

Amitanand Swaminathan Aiyer, B.Tech.; M.S.C.S.

DISSERTATION

Presented to the Faculty of the Graduate School of

The University of Texas at Austin

in Partial Fulfillment

of the Requirements

for the Degree of

DOCTOR OF PHILOSOPHY

THE UNIVERSITY OF TEXAS AT AUSTIN

August 2010

Dedicated to *Paati*,
my grandmother.

Acknowledgments

This work may not have been completed without the help of my parents, teachers, family, friends and colleagues who have helped, supported and encouraged me throughout the program.

I thank my advisor Prof. Lorenzo Alvisi for guiding and mentoring me and for consistently pushing me to get better at things that I had to improve upon, while ensuring that I enjoy what I do. I thank Prof. Rida Bazzi for the numerous brainstorming and technical discussions.

I thank all my committee members, Prof. Lorenzo Alvisi, Prof. Rida Bazzi, Prof. Mike Dahlin, Prof. Mohamed Gouda and Dr. Jay Wylie for their feedback and technical discussions on the various topics in this dissertation. I thank my colleagues: JP, Allen, Jeff, Taylor, Harry, Manos, Ed, Anurag, and Silvio for their collaboration and help. I thank my previous teachers and mentors: Ramaiah Sir, Koteshwar Sir, Prof. Pandu Rangan, Prof. Zuckerman, Sudipta, Jin, Li-wei, Xiaozhou, Mehul and Eric for helping me learn various tools and techniques that have been useful to my research.

I thank my parents, sister, aunts, uncles, cousins and all my friends for their continued support and encouragement. I thank my friends: Hetu, Nalini, Shiva, Meghana, Ajit, Prem, Murari, Karthik K, Ashwin, Aparna, Prakash, Lokesh, Shweta, Meghana, Harish, Shiva, Bakri, Balaji, Jagadish, Keerthi, Karthik M, Alok, Harsh, Vineet, Guneet, Nikita, Gill, Apurv, Akshay, Joy, Kriti, Anish, Sumala, Ramya, Sonal, Ruchica, Pooja, Pavithra, Piyush, Sibi,

Priyank, Upendra, Nishit, Ankit, Nisha, Monica, Lexi, Wendy, Sucharit, Ravi, Sunny and Mashal for their support and for making my stay in Austin so enjoyable.

Alternative implementations for storage and communication abstractions in distributed systems

Publication No. _____

Amitanand Swaminathan Aiyer, Ph.D.
The University of Texas at Austin, 2010

Supervisor: Lorenzo Alvisi

Abstractions are widely used in building reliable distributed systems as they simplifies the task of building complex systems and aid in reasoning about them. Implementing these abstractions, however, requires making certain assumptions about the environment in which they will be used.

We find that there is a mismatch in the set of assumptions used to implement abstractions in the different layers of a distributed system. This leads to a costlier design and may render the implementation unusable in situations where the assumptions do not hold.

In this dissertation we provide alternative implementations for the abstractions of distributed registers and communication channels that rely on a unified set of assumptions across the different layers of a distributed system.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

This dissertation revisits some of the fundamental abstractions behind implementing reliable distributed systems. Abstractions simplify the design and implementation of complex systems by identifying the properties provided by re-usable components in the system. This allows such components to be used across different systems saving development effort. It also allows for a simpler design and facilitates better reasoning by allowing the developer to build upon these higher-level components, that may provide stronger properties, than lower-level components which are used to implement them.

Ultimately however, to be used in building systems these abstractions must be implemented by making assumptions about the environment in which the implementation will be used. Making stronger assumptions about the environment allows us to implement an abstraction more efficiently; while making weaker assumptions allows the implementation to be applicable across a wider set of scenarios.

We find that there are many instances in distributed systems where abstractions are implemented using assumptions that different significantly from the assumptions that hold in the environment where the abstractions are used.

- In some cases, we find that certain abstractions are implemented using

fewer assumptions than what the environment allows for; thus, leading to costlier implementations.

- In some other cases, we find that existing implementations require assumptions that are too strong to hold in many practical situations where the corresponding abstractions might be useful.

Abstractions allow us to build reliable distributed systems in a layered manner. In this work, we view distributed systems as implementing a 3-layered architecture[1] as shown in Figure 1.1. The bottom *communication layer* masks

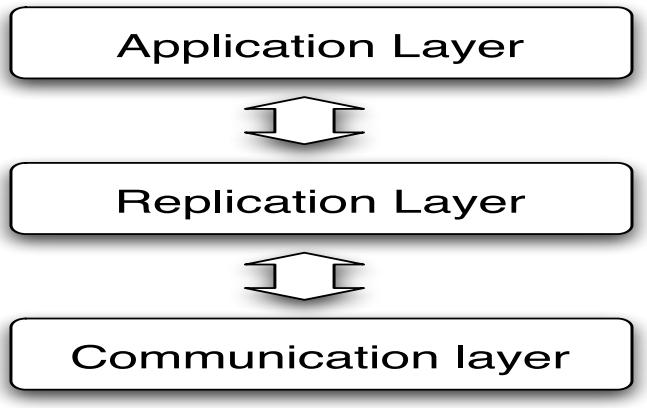


Figure 1.1: A three-layered architecture for distributed systems.

network failures and provides abstractions that allow processes to communicate securely with each other. The abstractions of authenticated channels and private channels, for example, are implemented in the communication layer. The middle *replication layer* builds upon the abstractions provided

[1]These layers can be further refined into more sub-layers. But, for the purpose of this work, we only focus on the abstractions at the interface between these three layers.

by the communication layer to replicate the processes' tasks. The replication layer masks process failures and provides higher-level abstractions. For example, distributed storage applications widely use the abstraction of a distributed register, which is implemented in the replication layer using quorum systems [5, 7, 17, 18, 32, 55, 66, 87, 90, 95, 96, 121]. Compute-centric applications use the abstraction of a single-reliable-node that can be implemented using state machine replication [1, 11, 29, 35, 36, 45, 48, 80, 103]. The top *application layer* uses the abstractions from the replication layer (distributed register, single-reliable-node etc.) to implement a distributed service or an application.

In this work, we focus on the implementations of distributed registers and the implementations of underlying communication abstractions where there is a mismatch in the assumptions between different layers:

First, we find that communication abstractions are commonly implemented based on assumptions that are weaker than the assumptions used in the replication layer. These communication abstractions are commonly implemented using cryptographic primitives that allow *any* number of faulty processes in the systems. The replication layer, on the other hand, provides meaningful guarantees only when the number of faulty processes in the system is bounded.

Second, we find that the existing implementations of the distributed register abstraction rely on assumptions that do not hold in the environments in which this abstraction is being used. Consider, for instance, large-scale storage applications spanning geographically distributed data centers. These applications need both high availability and the ability to tolerate network partitions, in the rare event where different data centers may not be able to communicate with each other. Unfortunately, existing quorum techniques that implement the distributed register abstractions provide only one of these two properties.

Strict quorum systems [15, 17, 52, 59, 61, 65, 89, 90, 91, 95, 94, 106, 118] can implement register abstractions that provide strong consistency guarantees in spite of an asynchronous network. However, unlike non-strict quorum systems such as probabilistic quorum systems [92] and signed quorum systems [121], strict quorum systems cannot provide high availability during transient failures. Probabilistic quorum systems and signed quorum systems provide higher availability at the cost of weakening the consistency guarantees provided, by allowing the register to sometimes return stale (old) values. If the network is synchronous, then these implementations can bound the probability of returning stale values. However, if the network may suffer from periods of asynchrony, these implementations can provide no guarantee on the staleness of the returned values. Thus, for a system that may deal with transient failures and may also be subject to network partitions, neither of the two approaches is satisfactory.

1.1 Outline and contributions

In this work, we address these problems by developing alternative implementations for communication abstractions and by proposing and implementing new register abstractions. The alternative implementations that we propose for communication abstractions utilize the additional assumptions present in the replication layer to replace the costly cryptographic primitives currently used in the communication layer with cheaper ones. The distributed register implementations we develop combine the salient features of existing distributed register implementations to provide high availability during periods of synchrony and withstand intermittent periods of asynchrony, ensuring a definite bound on the worst-case staleness.

Chapter 2 introduces the communication and register abstractions. Chapter 3 introduces the cryptographic primitives used in implementing various communication abstractions, and Chapters 4–6 provide the implementations. Chapter 7 introduces the quorum techniques used to implement distributed registers. Chapter 8 implements a novel register semantics, known as a k-atomic register, that can guarantee a definite bound on staleness despite asynchrony and provide higher availability when the system is synchronous.

We make the following contributions:

- We provide an alternative implementation of an authorized channel[2] that, under certain restrictive conditions, can be implemented using secret sharing techniques [23, 33, 113, 119] instead of digital signatures. Secret sharing techniques are computationally less expensive than digital signatures and also provide better security guarantees (Chapter 3). We discuss the techniques used to implement the authorized channel in Chapter 6.

- We provide an alternative implementation of a verifiable channel[2] that, under certain restrictive conditions, can be implemented using MACs instead of digital signatures. MACs are known to be computationally less expensive than digital signatures (Chapter 3). We discuss techniques used to implement the verifiable channel in Chapter 5.

- We provide alternative implementations for private channels and authenticated channels[2] among a group of n processes that, under restrictive conditions, use just $O(\log^2 n)$ keys at each process instead of $O(n)$ keys. We discuss these implementations in Chapter 3.

[2]Defined in Section 2.1

- We implement the k-atomic register which allows for a higher availability when the system is synchronous while ensuring a definite bound on staleness even under asynchrony.

Chapter 2

Architecture

In this chapter, we introduce the architecture we use for reasoning about reliable distributed systems.

Distributed systems consist of a set of *processes* that communicate with each other over a collection of *communication links*, known as the *network*; wherein both processes and the communication links may be subject to failures. Building a reliable distributed system is a complex task because it requires the system to be functional in spite of failed processes and communication links. A well-known technique to reduce the complexity of large systems is to build a layered architecture. A layered architecture implements a system as a series of layers one above the other. Each layer in the architecture provides a clear interface to the layers above and below it, while implementing a specific functionality.

A layered approach to system building provides several advantages: (i) It allows the designer to focus on relatively smaller problems in each layer; thus, simplifying the task of building and reasoning about the system. (ii) It allows the designers to potentially reuse the components in these layers across different systems; thus, saving development time and costs.

The number of layers in the architecture and the granularity of each layer is a subjective choice. For any given layering, it is always possible to imagine splitting each layer into multiple sub-layers or combining multiple

adjacent layers into a single one, depending on the anticipated level of reuse and reasoning. The ISO-OSI network stack model, for example, sub-divides the communication link between two processes into seven layers [116]. However, distributed systems that build over the communication links, without modifying the details of their implementation, are unaware of all but the top application layer.

In this dissertation, we view the architecture of distributed systems to consist of three layers as shown in Figure 1.1. The bottom layer is the *communication layer*: it allows processes to communicate with each other in a reliable and secure manner. The communication layer masks network faults and provides useful security guarantees for the messages received over the network. The middle layer is the *replication layer*: it masks process failures by replicating the state of the application across multiple processes. The replication layer builds upon the communication layer to provide higher-level abstractions for implementing reliable storage systems and for implementing general deterministic services, which may include storage and computation. The top layer is the *application layer*: it implement the desired application, over the replication layer, to provide the required levels of reliability and availability.

To understand this architecture in the context of a real system let us consider the example of Dynamo, which is used for implementing the shopping-cart application at Amazon [50]. For the shopping cart application to be functioning in a reliable manner Dynamo should function in spite of process failures and network losses. Moreover, given the scale at which Amazon operates, Dynamo must store many more key-value pairs than a single node can hold. The application layer in Dynamo consists of a distributed hash-table that stores all the key-value pairs in the system. Each key-value pair is stored in an abstract

register that is designed to tolerate process failures and network outages. To tolerate process failures, the replication layer implements the abstract register replicating the task of storing the data across multiple processes in the system. To store or retrieve the data, a designated leader process contacts a quorum of replicas to perform the desired operation. The communication layer provides abstractions to help the leader process communicate with the replicas in a secure manner.

2.1 The communication layer

The communication layer provides abstractions to facilitate communication between processes in a secure and reliable manner, inspite of network faults. The basic abstraction provided is that of a channel.

Definition. A channel $C(s, r)$ is a communication abstraction between two processes, a sender s and a receiver r, that implement the *send* and *deliver* methods. The *send* method allows the sender s to send a message on the channel. The *deliver* method allows the receiver r to receive a message that was sent on the channel; if there is no outstanding message to be received *deliver* blocks.

A process s is said to have *sent* a message msg (to r) on channel C if it has invoked the *send* method with the message msg on channel C. Similarly, a process r is said to have *delivered* a message msg (from s) on channel C if an invocation to the *deliver* method on the channel C returned msg. We denote by $Sent_{s,C}(\tau)$ (respectively, $Delivered_{r,C}(\tau)$) the set of messages that, at time τ, have been sent by s on (delivered by r from) the channel C. Both these sets grow monotonically with time.

2.1.1 Additional requirements

2.1.1.1 Timeliness

The definition of a channel does not specify anything about the timeliness or the order in which messages are delivered. If there is a known bound by which any message sent by the sender will be delivered at the receiver, the channel is said to be *synchronous*. Otherwise, it is said to be *asynchronous*.

2.1.1.2 Message loss

A channel may drop messages on the channel, reorder them, or modify them (as discussed below). A channel is said to be *FIFO* if messages are delivered by the receiver in the same order in which they are sent by the sender. A channel is said to be *reliable* if it does not drop messages. A channel is said to be *fair* if the channel may drop messages; but, a message sent an infinite number of times by the sender will be delivered by the receiver an infinite number of times. In an asynchronous network, reliable channels can be implemented over fair channels by retrying until the message is delivered.

2.1.2 Other security requirements

Untrusted networks Messages sent over a communication link [1] may be modified inadvertently because of factors, such as interference (e.g. wireless networks [108, 109]) or collision (e.g. Ethernet [98]). Malicious routers, in an untrusted network, could also modify messages. To protect against such scenarios we require additional properties for the channel. Privacy ensures that

[1] We use the term channel to denote an end-to-end message passing abstraction. Link is a term we use to refer to a physical communication medium between two neighboring processes that may be used to implement the channel.

a message sent over the channel cannot be eavesdropped upon by a malicious process. Authentication ensures that the receiver does not deliver any modified messages. We define communication channels that provide these properties:

2.1.2.1 Authenticated channel

Definition. An authenticated channel $C(s, r)$ guarantees that if the sender and receiver are correct, then the receiver r delivers a message msg only if the sender s sent the message.

$$\text{Authenticated channel} : msg \in Delivered_{r,C}(\tau) \Rightarrow msg \in Sent_{s,C}(\tau)$$

Authenticated channels are used in distributed systems to ensure that a malicious process cannot masquerade as the sender to send an arbitrary messages.

2.1.2.2 Private channel

Definition. A private channel $C(s, r)$ guarantees that if the sender and receiver are correct, then messages sent on the channel cannot be read by any other process.

Private channels are used to send confidential information. Bank transactions and other transactions that involve exchanging sensitive information are implemented over private channels.

Untrusted senders Authenticated and private channels protect the receiver in an untrusted network when the sender is correct. However, when the sender is malicious these channels do not protect the receiver. A malicious sender may,

for example, send unauthorized messages to corrupt the state at the receiver. Or, a malicious sender may also disown the messages it has sent to the receiver. Verifiable and authorized channels can protect the receiver in such scenarios.

2.1.2.3 Verifiable channel

A verifiable channel allows messages delivered through the channel to be attributed to the sender, such that not just the receiver but also other processes that do not directly receive the message from the sender can verify that the message was indeed sent on the channel. This property is known as verifiability. A k-verifiable channel provides verifiability for processes up to k hops from the original sender.

Definition. A k-verifiable channel $C_k(s, r_0)$ guarantees that

1. Whenever a correct receiver r_0 delivers a message msg, r_0 obtains a proof $\pi_1(s, msg)$ that is guaranteed to be accepted by any correct process r_1 as a proof that the sender s has sent the message msg.

2. For any chain of $k+1$ processes $r_0, r_1, r_2, r_3, \ldots r_k$, if a correct process r_i (for $i < k$) accepts $\pi_i(s, msg)$ as a proof that the sender s has sent the message msg on $C_k(s, r_0)$, then r_i can produce a proof $\pi_{i+1}(s, msg)$ that will be accepted by any correct process r_{i+1} as a proof that the sender s has sent the message msg on $C_k(s, r_0)$.

3. If the sender s is correct, a correct process r will not accept a proof $\pi(s, msg')$ for a message msg' that s has not sent on $C_k(s, r_0)$.

2.1.2.4 Authorized channel

Authorized channels protect a correct receiver from delivering unauthorized messages that a malicious sender may send.

Definition. Let $P(s, msg, \tau)$ be a predicate that evaluates to **true** only if the sender s is authorized to send the message msg by global-time τ. An authorized channel guarantees that a correct receiver r shall deliver a message msg from the channel $C(s, r)$, only if the sender is authorized to send the message.

$$\text{Authorized channel} : msg \in Delivered_{r,C}(\tau) \Rightarrow P(s, msg, \tau)$$

Byzantine fault tolerant systems have implicitly used the notion of an authorized channel to ensure that a correct receiver only delivers messages that the sender is authorized to send.

2.2 The replication layer

The replication layer builds upon the abstractions provided by the communication layer to handle process faults and provides higher-level abstractions that can be used by the application layer.

2.2.1 Process faults

A process is considered to be faulty if at any point during an execution, its behavior deviates from the specified protocol. A faulty process may deviate from the protocol in different ways.

1. *Crash:* A faulty process stops prematurely, and from there onwards it does not take any actions. Before stopping, however, it behaves correctly.

2. *Send omission:* A faulty process stops prematurely, or intermittently omits to send messages that it is supposed to send, or both.

3. *Receive omission:* A faulty process stops prematurely, or intermittently omits to receive messages that are sent to it, or both.

4. *General omission:* A faulty process is subject to send or receive ommission failures, or both.

5. *Byzantine:* A faulty process may exhibit any behavior whatsoever. It can make arbitrary state transitions and send any arbitrary message to other processes.

2.2.2 Abstractions in the replication layer

The replication layer provides abstractions that allow the application to replicate the application's state and services in a transparent manner. The abstraction of a state machine can be used to support any deterministic application using techniques such as primary-backup [30] and active replication [1, 11, 29, 35, 36, 45, 48, 80, 103]. For more specific applications, such as storage systems, the abstraction of a distributed register is used [5, 7, 17, 18, 32, 55, 66, 87, 90, 95, 96, 121]. In this thesis, we focus on the different abstractions of distributed registers that are implemented over a static set of processes i.e. one where the membership of the set does not change.

2.2.3 Register abstractions

The abstractions of a distributed register are useful to model the guarantees provided by storage systems. A register supports two operations: a write operation that allows one or more clients (writers) to update the value

stored in the register, and a read operation that allows one or more clients (readers) to access the value stored in the register. Each of these operations has a *start* time and an *end* time that are assigned using a global clock.

An operation O_A is said to *happen before* operation O_B ($O_A \rightarrow O_B$) if O_A ends before O_B starts. If neither $O_A \rightarrow O_B$, nor $O_B \rightarrow O_A$ then the operations O_A and O_B are said to be concurrent. A *serialized order* is a total order on the operations that is consistent with the partial order imposed by the *happens before* relation. A write operation O_w is said to be latest completed write to a read operation O_r if $O_w \rightarrow O_r$ and there is no other write operation $O_{w'}$ such that $O_{w'} \rightarrow O_r$ and $O_w \rightarrow O_{w'}$ [85, 93].

The consistency semantics of a register specifies the set of values that a read operation is allowed to return any under different scenarios. Lamport [85] defines three commonly-used consistency semantics of safe, regular, and atomic semantics for a single-writer system. These definitions have been extended to support multiple-writers as well [114]:

- **Safe register:** Any read operation that does not overlap with a write operation is guaranteed to return the result of latest completed write. The result of a read that overlaps with a write operation is unspecified.
- **Regular register:** Any read operation that does not overlap with a write operation is guaranteed to return the result of latest completed write. A read that overlaps with a write operation either returns the result of the last completed write, or one of the results of the write that it overlaps with.
- **Atomic register:** All read and write operations can be ordered in a total order that is consistent with the real time order, such that any read

operation returns the result of the last write operation that precedes the read operation in that order.

For systems that require higher availability and fault-tolerance, researchers have also considered weaker consistency semantics that relax the guarantees provided. Lee et al. have proposed the notion of a P-random register [87]:

- **P-random register:** A P-random register guarantees that a read operation will only return a value that was once written to the system. Moreover, the probability that a read operation returns a value that is more than l writes old is $P(l)$.

Distributed registers are implemented using quorum systems. We discuss these techniques in Chapter 7.

Chapter 3

Background: Cryptographic techniques

Secure communication abstractions are implemented using cryptographic primitives. In this chapter we introduce some of the commonly used cryptographic primitives and compare them with each other [62, 97].

3.1 Encryption

Encryption schemes enable processes to communicate privately with each other, over an untrusted network. An encryption scheme consists of two methods called Encrypt and Decrypt . The Encrypt method takes as input the message to be transmitted and outputs a string, known as the ciphertext or the encrypted message, that can be revealed to the adversary without giving away the contents of the message. The Decrypt method is used to retrieve the original message from the ciphertext. Both these methods are parameterized by keys which are kept secret. The encryption key (K_e) used to encrypt a message is known only to the sender(s); while the decryption key (K_d) used to decrypt the ciphertext is known only to the receiver(s). An encryption scheme is defined as follows:

Definition (Encryption scheme). An encryption scheme consists of the two

methods Encrypt and Decrypt.

$$Encrypt \quad : \quad \Sigma^* \times \Sigma^* \mapsto \Sigma^* \tag{3.1.1}$$

$$Decrypt \quad : \quad \Sigma^* \times \Sigma^* \mapsto \Sigma^* \tag{3.1.2}$$

The Encrypt method takes as input a message and the encryption key, and returns the ciphertext. The Decrypt method takes the decryption key and the ciphertext to return the decrypted message. These methods satisfy the following properties:

1. (Consistency) For all messages, msg,

$$Decrypt(K_d, Encrypt(K_e, msg)) = msg$$

2. (Confidentiality) It is impossible for an adversary to decrypt any message without the decryption key K_d.

Public/Private encryption. Encryption schemes are classified into two types, namely, public-key encryption and private-key encryption schemes. Private-key encryption schemes use the same *symmetric* key as both encryption key and decryption key. Thus, any process that can encrypt a message can also decrypt it. Such schemes are commonly used to implement private point-to-point channels. Public-key encryption schemes use different encryption and decryption keys, and thus allow for the set of processes that can encrypt a message to be different from the set of processes that can decrypt a message. These schemes are used to implement many-to-one channels and are also used for setting up private point-to-point channels.

3.2 Message authentication codes

Message authentication codes (MACs) allow processes to communicate with each other in an authenticated manner.

Definition (MAC scheme). A scheme for message authentication codes consists of two methods, MAC-Generate and MAC-Verify, parameterized by a common shared key:

$$\text{MAC-Generate} \quad : \quad \Sigma^* \times \Sigma^* \mapsto \Sigma^* \tag{3.2.1}$$

$$\text{MAC-Verify} \quad : \quad \Sigma^* \times \Sigma^* \times \Sigma^* \mapsto \textbf{Boolean} \tag{3.2.2}$$

The MAC-Generate method takes as input a message and the secret key, K_s, to return the message authentication code. The MAC-Verify method takes as input the secret key, K_s, the message and the message authentication code to verify weather the message authentication code is valid for the message. The verify method returns a boolean value indicating weather the message authentication code is valid for the message. These methods are required to satisfy the following properties:

1. (MAC-Consistency) For all messages, msg,

$$\text{MAC-Verify}(K_s, msg, \text{MAC-Generate}(K_s, msg)) = \textbf{true}$$

2. (MAC-Validity) It is impossible for an adversary who does not have access to the symmetric key K_s, to generate a message authentication code μ for a message msg, such that $\text{MAC-Verify}(K_s, msg, \mu) = \textbf{true}$.

3.3 Digital signatures

Signature schemes [63,110] allow data to be attributed to a process such that multiple processes can verify the claim. Thus, digital signature schemes

can be used to implement one-to-many communication schemes where a message generated by the sender can be authenticated by multiple verifiers, who can exchange the message among themselves. While, in general, digital signature schemes are allowed to be non-deterministic, commonly used digital signature schemes are deterministic. We define a deterministic digital signature scheme as follows:

Definition (Deterministic Digital Signature scheme). A deterministic digital signature scheme consists of two deterministic methods DS-Sign and DS-Verify, parameterized by a pair of private and public keys, respectively:

$$\text{DS-Sign} \quad : \quad \Sigma^* \times \Sigma^* \mapsto \Sigma^* \tag{3.3.1}$$

$$\text{DS-Verify} \quad : \quad \Sigma^* \times \Sigma^* \times \Sigma^* \mapsto \textbf{Boolean} \tag{3.3.2}$$

The DS-Sign method takes as input a message and the private key K_s, known only to the signer, and returns the signature. The DS-Verify method takes the public key, K_v, a message, a signature, and returns a boolean value indicating weather the signature is valid for the message. These methods are required to satisfy the following properties:

1. (DS-Consistency) For all messages, msg,

$$\text{DS-Verify}(K_v, msg, \text{DS-Sign}(K_s, msg)) = \textbf{true}$$

2. (DS-Validity) It is impossible for an adversary, who is given access to K_v but not K_s, to generate a signature σ for an unsigned message msg such that $\text{DS-Verify}(K_v, msg, \sigma) = \textbf{true}$.

3. (DS-Verifiability) If the verification method DS-Verify(K_v, msg, σ) evaluates to **true** once, then DS-Verify(K_v, msg, σ) evaluates to **true** always, regardless of which process invokes DS-Verify.

If the verification method DS-Verify is deterministic, then the scheme satisfies DS-Verifiability. We prefer to state this property explicitly because DS-Verifiability is crucial in achieving verifiability. This property allows one process, say r_1, who has verified that a message was signed by s, to forward the message and the signature to a different process r_2, who has K_v, to convince r_2 that the message was signed by s.

3.4 Secret sharing

Secret sharing [23,33,113,119] primitives allow a secret S to be split into multiple shares, $Sh_1, Sh_2, \ldots, Sh_n$, such that the secret can be reconstructed if and only if at least k such shares are pooled together. Specifically, a (k, n)-secret sharing scheme provides the following guarantees:

1. Knowledge of k or more shares from $Sh_1, Sh_2, \ldots, Sh_n$ makes the secret S easily computable.

2. Knowledge of any $k-1$ or fewer shares from $Sh_1, Sh_2, \ldots, Sh_n$ leaves the secret S completely undetermined (i.e. it could take any possible value).

Property (2) essentially differentiates secret sharing from erasure coding [32, 40, 72, 73], where receiving fewer than k shares might provide partial information. We later show in Chapter 6 how secret sharing can be used to implement certain authorized channels.

3.5 The costs involved

Cryptographic primitives provide strong security properties that are very useful. However, the use of cryptographic primitives has various costs associated with it. We consider three such costs:

1. The computational costs required to compute the primitives. These computational costs are incurred each time the cryptographic primitive is used. Thus, for efficiency reasons distributed systems implementations may choose using cryptographic primitives that are less costly.

   Implementations of message authentication codes (MACs), for example, are known to be 2-3 orders of magnitude faster than implementations of digital signatures [34]. Thus, several systems try to restrict the use of digital signatures and use MACs instead to achieve better performance [34, 37, 48, 80].

2. The storage costs needed to set up the required infrastructure to enable using these cryptographic primitives. Cryptographic primitives commonly require the use of secret keys shared between two or more communicating parties. The amount of storage space needed to store these keys is an important consideration in practice.

   Implementations of encryption schemes and authentication schemes based on one-time pads are stronger than the implementations based on symmetric keys. However, using one-time pads requires storage in the same order of magnitude as the total size of messages exchanged over the lifetime of the system. This is a very high cost: thus, in practice systems use symmetric keys instead of one-time pads.

3. The set of additional assumptions required on the adversary. Every assumption on the adversary that is required to implement a primitive restricts the use of the primitive because, if the assumption does not hold, then the cryptographic primitives based on such an assumptions will be vulnerable and not provide any useful guarantee. Public-key encryption schemes, for example, require that the adversary be computationally bounded. If the adversary is not computationally bounded, then the adversary can guess the secret keys, and the primitives do not provide the guarantees they should. Symmetric encryption based on one-time pads or secret sharing primitives do not require such additional assumptions and can be deployed even when the adversary has unlimited computational resources.

   A cryptographic scheme that provides the required guarantees when the adversary only has bounded computational resources is said to be *computationally secure*. A cryptographic scheme that provides the required guarantees even when the adversary has unbounded computational resources is said to be *unconditionally secure*.

Chapter 4

Authenticated and private channels

In this chapter, we discuss the implementation of authenticated channels and private channels. Authenticated and private channels are useful to ensure that the communication between the sender and the receiver is secure against other processes. An authenticated channel ensures that outside processes cannot inject messages into the channel; while a private channel ensures that outside processes cannot eavesdrop on the messages sent over the channel.

We first revisit the definitions for these channels and provide an implementation for authenticated and private channels, respectively in Section 4.1 and 4.2. We then discuss in Section 4.3 protocols to implement these channels among a group of processes such that any two process can communicate with each other.

4.1 Authenticated channels

Definition. An authenticated channel $C(s, r)$ guarantees that if the sender and receiver are correct, then the receiver r delivers a message msg only if the sender s sent the message.

$$\text{Authenticated channel : } msg \in Delivered_{r,C}(\tau) \Rightarrow msg \in Sent_{s,C}(\tau)$$

4.1.1 Implementation

Authenticated channels can be implemented using message authentication codes (MACs)[1]. To implement an authenticated channel, the sender s and the receiver r share a secret key, K_{auth} (known only to the two processes s and r) that is used to generate and verify message authentication codes. Figure 4.1 shows an implementation of the authenticated channel.

```
AuthenticatedChannel extends Channel {
    Authenticated-Send(Message msg) {
        h = MAC-Generate(K_auth, msg);
        Channel-Send(⟨msg, h⟩);
    }

    Message Authenticated-Deliver() {
        do {
            ⟨msg, h⟩ = Channel-Deliver();
            if (MAC-Verify(K_auth, msg, h) = true})
                return msg;
        } while (true);
    }
}
```

Figure 4.1: Implementation: Authenticated channel.

Theorem 1. The algorithm in Figure 4.1 implements an authenticated channel.

Proof. If both the sender and the receiver are correct, then the secret key K_{auth} is not revealed. Since a correct receiver delivers a message only if it is accompanied by a valid MAC, it follows from the MAC-Validity property of the underlying message authentication scheme (Definition 3.2) that the message must have been sent to the receiver, by either the receiver itself, or by the sender s. Since the receiver is correct, the sender must have sent the message. □

[1] Authenticated channels can also be implemented using digital signatures; however, message authentication codes are preferred because they are computationally less expensive.

4.2 Private channels

Definition. A private channel $C(s, r)$ guarantees that if the sender and receiver are correct, then messages sent on the channel cannot be read by any other process.

4.2.1 Implementation

Private channels are commonly implemented using symmetric encryption schemes[2]. To implement a private channel, $C(s, r)$, the sender s and the receiver r share a common secret key K_{enc} that is used to encrypt and decrypt messages.

```
PrivateChannel extends Channel{
    Private-Send(Message msg) {
        encrypted_message = Encrypt(K_enc, msg);
        Channel-Send(encrypted_message);
    }

    Message Private-Deliver(msg) {
        encrypted_message = Channel-Deliver();
        original_message = Decrypt(K_dec, msg);
        return original_message;
    }
}
```

Figure 4.2: Implementation: Private channel.

Theorem 2. The algorithm in Figure 4.2 implements a private channel.

Proof. If both the sender and the receiver are correct, then the secret key K_{enc} is not revealed to any other process. It follows from the underlying encryption scheme's confidentiality requirement (Definition 3.1) that no process, other than s and r, can read the messages sent on the channel. □

[2] Private channels can also be implemented using public key encryption. However, public key encryption is computationally more expensive than symmetric key encryption.

4.3 Communication in a group

To implement an authenticated channel or a private channel between a pair of processes, the sender and the receiver are required to share a symmetric key that is not known to any other process. Thus, implementing authenticated channels or private channels among a group of n processes requires assigning a secret key to each of the $\binom{n}{2}$ channels such that the symmetric key assigned to channel $C_{A,B}$ is known only to processes A and B. Current implementations generate these keys independently: thus, implementing authenticated channels or private channels among a group of n processes requires generating a total of $O(n^2)$ keys, where each process is required to store the $O(n)$ keys that it uses for communication.

For settings such as sensor networks [54,104], ad-hoc networks [78,117], and mobile networks [79,120], where the amount of computational and storage resources available to each process is limited, storing $O(n)$ keys at each process may not be desirable. Using solutions based on public-key encryption or digital signatures may be computationally expensive. We explore techniques aimed at reducing the number of symmetric keys to be stored by each process.

The problem of reducing the number of keys stored at each process has been studied before [64,83]. It is known that under more restrictive conditions, the number of keys that each process needs to store can be drastically reduced. If processes do not collude with each other, then the symmetric keys used for communication need not be independent of each other: it is enough to ensure that no process – by itself – can generate the keys that two other processes use for communication. Gong and Wheeler [64] use this observation to develop a scheme that allows n processes to communicate with each other while storing just $O(\sqrt{n})$ keys at each process.

We improve upon their result to present a family of protocols that allows processes to communicate with each other using just $O(\sqrt[k]{n})$ keys for any constant k.

Section 4.3.1 presents the protocol by Gong and Wheeler [64]. Section 4.3.2 presents our contribution, which is a family of key distribution protocols that generalizes Gong and Wheeler's technique to reduce the number of keys to $O(k^2 \sqrt[k]{n})$ keys for any $1 \leq k \leq \log n$. In the extreme case, when $k = \log n$ this results in a scheme that assigns only $O(\log^2 n)$ keys to each process. Section 4.3.3 presents a lower bound on the number of keys required, and Section 4.3.4 discusses the related work.

4.3.1 The Basic Grid Protocol

Model Consider a set of n processes. Each process in the network has a unique identifier in the range $0 \ldots n-1$, represented by $\log n$ bits. The processes need to be able to communicate among each other securely (i.e. over private or authenticated channels) using as few keys as possible.

Construction We partition the $\log n$ bits of the process identifiers into two groups, called A-bits and B-bits. The A-bits can have $umax$ distinct values, 0 through $umax - 1$, and the B-bits can have $vmax$ distinct values, 0 through $vmax - 1$. We require the two groups to be as close as possible to equally sized, so that both $umax$ and $vmax$ are $O(\sqrt{n})$.

Each process is mapped to a point on a $umax \times vmax$ grid. An element (u, v) in this grid corresponds to the process whose A-bits has the value u and whose B-bits has the value v. We refer to this process as $p(u, v)$.

Key assignment We specify two types of symmetric keys, called grid keys and direct keys, for the different elements in the grid, according to the following two rules.

i. For each grid element (u, v), specify a random *grid key* denoted $g(u, v)$.

ii. For each pair of grid elements (u, v) and (u', v'), where $u = u'$ or $v = v'$, specify a random *direct key* denoted $d(u, v)(u', v')$. Note that the direct key $d(u, v)(u', v')$ can also be denoted by $d(u', v')(u, v)$, since the order of the two pairs (u, v) and (u', v') is immaterial in the name of a direct key.

The specified grid and direct keys are assigned to the system processes as follows.

a. Each process $p(u, v)$ is assigned a copy of every grid key of the form $g(u, v')$ and a copy of every grid key of the form $g(u', v)$. Thus, each process is assigned approximately $(umax + vmax)$ grid keys.

b. Each process $p(u, v)$ is also assigned a copy of every direct key of the form $d(u, v)(u, v')$ and a copy of every direct key of the form $d(u, v)(u', v)$. Thus, each process is assigned approximately $(umax+vmax)$ direct keys.

It follows that the total number of keys assigned to each process is $2(umax + vmax)$ keys. Since each of $umax$ and $vmax$ is $O(\sqrt{n})$, the total number of keys assigned to each process is $O(4\sqrt{n})$.

Computing symmetric keys When a process $p(u, v)$ needs to communicate securely with another process $p(u', v')$, $p(u, v)$ uses (u, v) and (u', v') to

```
1.  Initially, SK is empty.

2.  if u ≠ u' and v ≠ v' →
            add the two grid keys, g(u,v') and g(u',v) to SK
      [] u ≠ u' and v = v' →
            add the direct key d(u,v)(u',v) to SK
      [] u = u' and v ≠ v' →
            add the direct key d(u,v)(u,v') to SK
      [] u = u' and v = v' → / impossible
            skip
    fi
```

Figure 4.3: Algorithm 1 for selecting keys.

compute a non-empty subset SK of its own keys that satisfies the following two conditions.

1. *Sharing:*

   Each key in the computed subset SK is assigned to both $p(u, v)$ and $p(u', v')$.

2. *Exclusion:*

   No process, other than $p(u, v)$ and $p(u', v')$ is assigned all the keys in the computed subset SK.

After computing the subset SK, process $p(u, v)$ applies an exclusive-OR on the keys in SK in order to compute a single shared key that $p(u, v)$ can use to communicate securely with $p(u', v')$. Process $p(u', v')$ also computes the same subset SK and applies an exclusive-OR on the keys in SK in order to compute a single shared key that $p(u', v')$ can use to communicate securely with $p(u, v)$.

The algorithm that each of $p(u, v)$ and $p(u', v')$ use to compute the subset SK is shown in Figure 4.3.

Theorem 3. The key subset SK computed by Algorithm 1 satisfies the two conditions of sharing and exclusion.

Proof. Assume that process $p(u, v)$ needs to communicate securely with process $p(u', v')$ and so it uses Algorithm 1 to compute the set SK of shared keys between $p(u, v)$ and $p(u', v')$. There are three cases to consider:

1. ($u \neq u'$ and $v \neq v'$): In this case, $SK = \{g(u, v'), g(u', v)\}$. Both $p(u, v)$ and $p(u', v')$ are assigned the two grid keys in SK and no other process is assigned both these keys. Thus, the computed SK satisfies the two conditions of sharing and exclusion.

2. ($u \neq u'$ and $v = v'$): In this case, $SK = \{d(u, v)(u', v)\}$. Both $p(u, v)$ and $p(u', v')$ are assigned the direct key in SK and no other process is assigned this key. Thus, the computed SK satisfies the two conditions of sharing and exclusion.

3. ($u = u'$ and $v \neq v'$): In this case, $SK = \{d(u, v)(u, v')\}$. Both $p(u, v)$ and $p(u', v')$ are assigned the direct key in SK and no other process is assigned this key. Thus, the computed SK satisfies the two conditions of sharing and exclusion.

□

4.3.2 A General Multi-Grid Protocol

We now generalize the basic grid protocol [64] to build a family of protocols that allow n processes to communicate with each other by storing just $O(k^2 \sqrt[k]{n})$ keys for any $1 \leq k \leq \log n$.

Construction Processes are represented by unique identifiers from 0 to $(n-1)$ using $\log n$ bits. These $\log n$ bits are partitioned into k parts – A_0-bits, A_1-bits,... A_{k-1}-bits. For every $i, 0 \leq i < k$, the A_i-bits have $umax_i$ distinct values, 0 through $umax_i - 1$. We want the number of bits in each part to be as equal as possible: for any two parts A_i-bits and A_j-bits, $|\log umax_i - \log umax_j|1$. Note that each $umax_i$ is $O(\sqrt[k]{n})$.

The processes are mapped on to a k-dimensional hypercube of sides $umax_0 \times umax_1 \times \ldots \times umax_{k-1}$. This k-dimensional hypercube is then represented using $\binom{k}{2}$ 2-dimensional projections as demonstrated in Figure 4.4. The two-dimensional grid consisting of dimensions i and j (for each $0 \leq i < j \leq k-1$) is called the A_{ij}-grid, and has $umax_i \times umax_j$ elements.

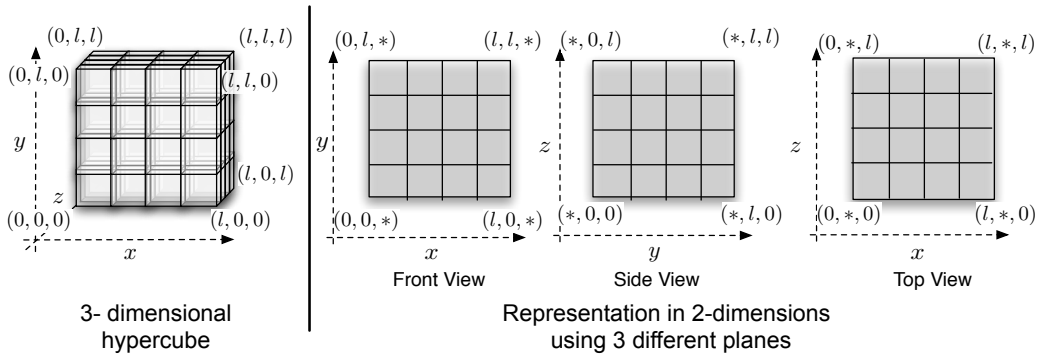


Figure 4.4: Representing a 3-dimensional hyper-cube in 2 dimensions.

Each element (u_i, u_j) in an A_{ij}-grid corresponds to the set of all processes where the A_i-bits have the value u_i and where the A_j-bits have the value u_j. Each process $p(u_0, u_1, \ldots, u_{k-1})$ corresponds to $\binom{k}{2}$ elements: element (u_0, u_1) in the A_{01}-grid, element (u_0, u_2) in the A_{02}-grid, ..., and element (u_{k-2}, u_{k-1}) in the $A_{(k-2)(k-1)}$-grid.

Key assignment Grid and direct keys are specified for the elements of every A_{ij}-grid according to the following two rules.

i. For each element (u_i, u_j) in the A_{ij}-grid, specify a random *grid key* denoted A_{ij}-$g(u_i, u_j)$.

ii. For each pair of elements (u_i, u_j) and (u'_i, u'_j), where $u_i = u'_i$ or $u_j = u'_j$ specify a random *direct key* denoted A_{ij}-$d(u_i, u_j)(u'_i, u'_j)$.

The specified grid and direct keys are assigned to the system processes as follows.

a. Each process $p(u_0, \ldots, u_{k-1})$ is assigned a copy of every grid key of one of the following two forms: A_{ij}-$g(u_i, u'_j)$ and A_{ij}-$g(u'_i, u_j)$. Thus each process is assigned $(k-1)(umax_0 + umax_1 + \ldots + umax_{k-1})$ grid keys.

b. Each process $p(u_0, \ldots, u_{k-1})$ is assigned a copy of every direct key of one of the following two forms: A_{ij}-$d(u_i, u_j)(u_i, u'_j)$ and A_{ij}-$d(u_i, u_j)(u'_i, u_j)$. Thus, each process is assigned $(k-1)(umax_0 + umax_1 + \ldots + umax_{k-1})$ direct keys.

Since each $umax_i$ is $O(\sqrt[k]{n})$, the total number of keys assigned to each process is $O(2k(k-1)\sqrt[k]{n})$.

Computing symmetric keys When a process $p(u_0, \ldots, u_{k-1})$ needs to communicate securely with another process $p(u'_0, \ldots, u'_{k-1})$, $p(u_0, \ldots, u_{k-1})$ computes a non-empty subset SK of its own keys that satisfies the following two conditions.

```
1.  Initially, SK is empty.

2.  For each A_ij-grid do
        if u_i ≠ u'_i and u_j ≠ u'_j →
                add the two grid keys, A_ij-g(u_i, u'_j) and A_ij-g(u'_i, u_j) to SK
          [] u_i ≠ u'_i and u_j = u'_j →
                add the direct key A_ij-d(u_i, u_j)(u'_i, u_j) to SK
          [] u_i = u'_i and u_j ≠ u'_j →
                add the direct key A_ij-d(u_i, u_j)(u_i, u'_j) to SK
          [] u_i = u'_i and u_j = u'_j →
                add the direct key A_ij-d(u_i, u_j)(u_i, u_j) to SK
        fi
```

Figure 4.5: Algorithm 2 for selecting keys.

1. *Sharing:*

   Each key in SK is assigned to both $p(u_0, \ldots, u_{k-1})$ and $p(u'_0, \ldots, u'_{k-1})$

2. *Exclusion:*

   No process other than $p(u_0, \ldots, u_{k-1})$ and $p(u'_0, \ldots, u'_{k-1})$ is assigned all the keys in SK.

After computing SK, process $p(u_0, \ldots, u_{k-1})$ applies an exclusive-OR on the keys in SK in order to compute a single shared key that $p(u_0, \ldots, u_{k-1})$ can use to communicate securely with $p(u'_0, \ldots, u'_{k-1})$. Process $p(u'_0, \ldots, u'_{k-1})$ also computes the same subset SK and applies an exclusive-OR on the keys in SK in order to compute a single shared key that $p(u'_0, \ldots, u'_{k-1})$ can use to communicate securely with $p(u_0, \ldots, u_{k-1})$.

The algorithm that each of $p(u_0, \ldots, u_{k-1})$ and $p(u'_0, \ldots, u'_{k-1})$ use to compute the subset SK is shown in Figure 4.5.

The number of keys in the computed subset SK depends on $(u_0, \ldots, u_{k-1})$ and $(u'_0, \ldots, u'_{k-1})$. For example, SK has the maximum number of keys, $k(k-1)$, when $u_i \neq u'_i$ for every $i = 0, \ldots, k-1$.

Theorem 4. The key subset SK computed by Algorithm 2 satisfies the two conditions of sharing and exclusion.

Proof. Assume that process $p(u_0, \ldots, u_{k-1})$ needs to communicate securely with process $p(u'_0, \ldots, u'_{k-1})$ and so it uses Algorithm 2 to compute set SK of shared keys between the two. It can be seen from Figure 4.5 that keys added to SK are assigned by both the communicating processes. Thus, Algorithm 2 satisfies the sharing condition.

To prove exclusion, we argue that no process other than $p(u_0, \ldots, u_{k-1})$ and $p(u'_0, \ldots, u'_{k-1})$ is assigned all the keys in SK. Let $p(v_0, \ldots, v_{k-1})$ be a process that is different from $p(u_0, \ldots, u_{k-1})$ and $p(u'_0, \ldots, u'_{k-1})$. There exists indices $0 \leq i, j \leq (k-1)$:

$$u_i \neq v_i \text{ and } u'_j \neq v_j$$

There are two cases to consider:

- $(i \neq j)$: Consider the keys assigned from the A_{ij}-grid: Since (v_i, v_j) is different from both (u_i, u_j) and (u'_i, u'_j), $p(v_0, \ldots, v_{k-1})$ is not assigned the grid keys that A_{ij}-grid contributes to SK.

- $(i = j)$: Consider the keys assigned from the $A_{i,i+1 \bmod k}$-grid: Since $(v_i, v_{i+1 \bmod k})$ is different from both $(u_i, u_{i+1 \bmod k})$ and $(u'_i, u'_{i+1 \bmod k})$, $p(v_0, \ldots, v_{k-1})$ is not assigned the keys that $A_{i,i+1 \bmod k}$-grid contributes to SK.

□

4.3.3 Lower bound

If processes do not collude with each other, we have shown that n processes can communicate securely with each other using $O(\log^2 n)$ keys. We now show a tight lower bound that each process needs to store at least $O(\log n)$ keys.

Theorem 5. In a a network of n processes, each process needs to be assigned at least $O(\log n)$ symmetric keys in order that each process is able to communicate securely with every other process in the network.

Proof. Assume that each process p in this network is assigned x keys. Process p needs to use a different non-empty subset of its x keys to communicate securely with every other process in the network. Because there are $2^x - 1$ non-empty subsets of the set of x keys, and process p needs to communicate securely with $(n-1)$ processes, we have

$$\begin{aligned} & 2^x - 1 \geq n - 1 \\ \Rightarrow \quad & x \geq \log n \end{aligned}$$

□

Theorem 5 establishes a lower bound on the number of keys that each process should store in order to be able to communicate securely with each of the other $(n-1)$ processes. We now show that this bound is tight by proving that it is possible to assign $O(\log n)$ keys to each of the n process so that they can communicate securely with each other.

To show that such a key distribution scheme exists, we use the probabilistic method. The probabilistic method [14] is a non-constructive proof

technique that is useful for showing that among a family of elements (here, key assignment protocols) there exists an element that satisfies a certain desired property. To prove the existence of such an element, we derive an upper bound on the probability that a randomly chosen element from the family does not satisfy the desired property. If we can show that this upper bound is strictly less than 1, we have proven that there exists an element in the family that satisfies the desired property.

Theorem 6. Given a network of n processes and a set of $12 \log n$ distinct keys, there is a protocol for assigning $9 \log n$ distinct keys from the given set of keys to each process in the network such that each process can use its assigned keys to communicate securely with each other process in the network.

Proof. Consider a family of key distribution protocols, where $12 \log n$ distinct keys are to be randomly distributed among n processes such that each process is assigned $9 \log n$ keys. In this case, any two distinct processes have at least $6 \log n$ keys in common.

Because the keys are assigned randomly to the network processes, the resulting key assignment can be either secure or insecure. In what follows we show that the probability that the resulting key assignment is insecure is strictly less than one. This implies that the probability that the resulting key assignment is secure is strictly more than zero.

Let $\mathcal{P}_{in}$ be the probability that a randomly generated key assignment from the family is insecure. If a key assignment is insecure there exists three distinct processes p, p' and p'' such that all the keys that are assigned to both p and p' (and so can be used by p and p' to communicate securely with one

another) are also assigned to p''. Thus,

$$\mathcal{P}_{in} \leq \binom{n}{2}(n-2)\frac{X}{Y}$$

$$< \frac{n^3}{2}\frac{X}{Y}$$

Note that the factor $\binom{n}{2}$ is the number of choices of the distinct processes p and p' from the set of n processes in the network; the factor $(n-2)$ is the number of choices of process p'' that is distinct from both p and p'. X is the number of ways of assigning $9\log n$ keys to process p'' under the assumption that $6\log n$ of those keys are those shared keys between p and p''; and Y is the number of ways of assigning $9\log n$ keys to process p''.

$$X = \binom{6\log n}{3\log n} \text{ and } Y = \binom{12\log n}{9\log n}$$

Therefore, the probability $\mathcal{P}_{in}$ that the randomly generated key assignment is insecure is:

$$\begin{aligned}
\mathcal{P}_{in} &< \frac{n^3}{2}\frac{\binom{6\log n}{3\log n}}{\binom{12\log n}{9\log n}} \\
&= \frac{n^3}{2}\frac{(6\log n)(6\log n-1)\ldots(3\log n+1)}{(12\log n)(12\log n-1)\ldots(9\log n+1)} \\
&< \frac{n^3}{2}\frac{(6\log n)(6\log n)\ldots(6\log n)}{\underbrace{(12\log n)(12\log n)\ldots(12\log n)}_{3\log n \text{ times}}} \\
&= \frac{n^3}{2}(\frac{1}{2})^{3\log n} \qquad = \frac{n^3}{2}(\frac{1}{2^{3\log n}}) \\
&= \frac{n^3}{2}(\frac{1}{n^3}) \qquad = \frac{1}{2} \qquad < 1
\end{aligned}$$

□

4.3.4 Related work

Gong and Wheeler [64] proposed the basic grid protocol to enable n processes to communicate with each other using $O(\sqrt{n})$ keys. Kulkarni et al. have proved the optimality of this result, assuming that two processes combine no more than two shared keys to generate a unique key used for communication [83].

In our work, we do not bound the number of shared keys combined to generate the unique key. Thus, we are able to show the existence of a key distribution protocol that assigns only $O(\log n)$ keys to each process. We have further shown that this result is asymptotically optimal.

The family of keys presented in Section 4.3.2 can construct a key distribution protocol that assigns to each process only $O(\log^2 n)$ keys. Following our work, Elmallah et al. [53] have explicitly constructed a key distribution protocol assigning only $O(\log n)$ keys to each process, which is optimal.

Kulkarni and Bezawada [82] consider the scenario in which processes may collude with each other and define metrics that quantify a key distribution protocol's vulnerability to collusion among processes.

Chapter 5

The verifiable channel

In this chapter we discuss the verifiable channel and its implementations. First, we revisit the definition in Section 5.1. In Section 5.2, we present an implementation based on digital signatures. Later, in Sections 5.3–5.5 we present implementations based on message authentication codes.

5.1 Verifiable channels

Verifiable channels are stronger than authenticated channels and provide verifiability in addition to authentication. This property is useful in Byzantine fault tolerant systems, where a faulty process could send mutually inconsistent messages to different processes. Moreover, a faulty process may also falsely implicate another process by claiming to have received a message that was not sent. Verifiability helps in ensuring that a process is held responsible only for messages that it has sent. The k-verifiable channel, defined in Chapter 2, provides verifiability up to k hops. As we are dedicating the rest of this chapter to an efficient implementation of the verifiable channel, for convenience we repeat the definition below:

Definition. A k-verifiable channel $C_k(s, r_0)$ guarantees that

1. Whenever a correct receiver r_0 delivers a message msg, r_0 obtains a proof $\pi_1(s, msg)$ that is guaranteed to be accepted by any correct process r_1

as a proof that the sender s has sent the message msg.

2. For any chain of $k+1$ processes $r_0, r_1, r_2, r_3, \ldots r_k$, if a correct process r_i (for $i < k$) accepts $\pi_i(s, msg)$ as a proof that the sender s has sent the message msg on $C_k(s, r_0)$, then r_i can produce a proof $\pi_{i+1}(s, msg)$ which will be accepted by any correct process r_{i+1} as a proof that the sender s has sent the message msg on $C_k(s, r_0)$.

3. If the sender s is correct, a correct process r will not accept a proof $\pi(s, msg')$ for a message msg' that s has not sent on $C_k(s, r_0)$.

Broadcast channels The abstraction of a broadcast channel has also been widely used for building reliable distributed systems [25, 26, 70, 84, 112]. A reliable broadcast channel among a set of processes guarantees that a message sent by a sender will be delivered by all correct processes; further, if a correct process r_i delivers a message msg from s, then every correct process r_j in the set will deliver the message msg from s. Thus, the properties provided by a verifiable channel are closely related to the properties provided by a reliable broadcast channel [26, 49, 70]. With a verifiable channe, the process r_{i+1} is convinced that a message was sent on the channel by the sender s – in a lazy manner – as needed, when r_{i+1} receives the proof $\pi(s, msg)$ from r_i; an implementation using a broadcast channel would ensure that all correct processes deliver the message msg from s, regardless of whether it is used. Pease et al. [103] show that under synchronous conditions a reliable broadcast channel can be implemented using an ∞-verifiable channels. Srikanth and Toueg [115] show that a broadcast channel can be implemented over authenticated point-to-point channels if the number of Byzantine faulty processes in the system is less than one-third.

5.2 An implementation using digital signatures

Verifiable channels are commonly implemented using digital signatures. Figure 5.1 shows an implementation of an ∞-verifiable channel. The sender

```
VerifiableChannel extends Channel {
    // Sender s, with private/public key pair (K_s, K_v)
    Verifiable-Send(Message msg) {
        σ_msg = DS-Sign(K_s, msg);
        Channel-Send(⟨msg, σ_msg⟩);
    }

    // receiver r_1
    (Message, proof) Verifiable-Deliver() {
        do {
            ⟨msg, σ⟩ = Channel-Deliver();
            if (DS-Verify(K_v, msg, σ) = true)
                return (msg, σ);
        } while (true);
    }

    // process r_i for i ≥ 1
    (Boolean, Signature) SentByTheSender( msg, proof ) {
        if (DS-Verify(K_v, msg, proof) = (true, new_proof) )
            return (true, new_proof);
        return (false, *);
    }
}
```

Figure 5.1: Implementation: Verifiable channel.

s digitally signs every message msg that is sent on the channel $C(s, r_0)$. To deliver a message, receiver r_0 verifies that the digital signature on the message is from the sender s and then delivers the message only if it is signature is correct. This convinces the receiver that the sender must have sent the message. Now, any correct process r_i that has verified the signature can send the signature, as a proof, to convince any other correct process r_{i+1} that the message was sent by the sender s.

Theorem 7. The algorithm in Figure 5.1 implements an ∞-verifiable channel.

Proof. A correct process r accepts that a message msg was sent by the sender s if and only if it is accompanied by a valid signature σ_{msg} from s. Thus,

1. A correct receiver r_0 delivers a message msg only if the accompanied by a valid digital signature σ_{msg} for the message. The signature σ_{msg} serves as a proof that will be accepted by any correct process r_1.

2. Any correct process r_i that accepts σ_{msg} as a proof, will only do so after verifying that DS-Verify$(K_v, msg, \sigma_{msg}) =$ **true**. DS-Verifiability guarantees that r_i can use σ_{msg} as a proof $\pi_{i+1}(s, msg)$ that will be accepted by any correct process r_{i+1}.

3. If the sender s is correct, then the private key K_s is not revealed to any other process. Thus, from DS-Validity, it follows that a correct process r will not accept a proof $\pi(s, msg')$ for a message msg' that s has not sent.

□

Unfortunately, digital signatures are not cheap. In particular, digital signature are two to three orders of magnitude more expensive to generate and verify that message authentication codes [34]. The rest of this chapter is dedicated to a very natural question: is it possible to use MACS to achieve the same properties of digital signatures? If so, it would be possible to achieve dramatically more efficient implementations of verifiable channels.

5.3 A MAC-based implementation

Although in general it is impossible to use MACs to implement digital signatures, we show that under certain circumstances it is possible to achieve

the properties of digital signatures, and hence implement k-verifiable channels, using message authentication codes.

Model We assume that the system consists of two sets of processes: a set of n server processes and a finite set of client processes (signers and verifiers). Less than one-third of the n servers may be faulty; and faulty servers may behave arbitrarily. Clients communicate with the servers over authenticated point-to-point channels. Inter-server communication is not required. The network is asynchronous and fair—but, for simplicity, our algorithms are described in terms of reliable FIFO channels.

5.4 Deterministic digital signatures

We now describe constructions that use MACs to provide the same properties as digital signatures [10]. These constructions can thus replace digital signatures from the implementation in Figure 5.1 to develop an alternative implementation that only uses MACs.

5.4.1 The high-level idea

We first present the high-level idea assuming two trusted entities in the system. One trusted entity acts as a signing witness and one acts as a verifying witness. The two witnesses share a secret-key K which is used to generate and verify MACs.

Signing a message A signer delegates to the signing witness the task of signing a message. This signing witness generates, using the secret key K, a MAC value for the message m to be signed and sends the MAC value to the

signer. This *MAC-signature* certifies that the signer s wants to sign m. It can be presented by a verifier to the verifying witness to validate that s has signed m.

Verifying a signature To verify that a MAC-signature is valid, a verifier (client) delegates the verification task to the verifying witness. The verifying witness computes, using the secret key K, the MAC for the message and verifies that it is equal to the MAC presented by the verifier. If it is, the signature is accepted; otherwise, it is rejected.

Since the two witnesses are trusted and only they know the secret key K, this scheme satisfies *consistency*, *validity*, and *verifiability*.

5.4.2 Implementing deterministic digital signatures with MACs

We now show that when $n > 3f$, MACs can be used to implement a deterministic digital signature scheme.

Construction In our signature scheme, the signatures produced are vectors of $N = \binom{n}{2f+1}$ MAC values, one for each subset of $2f + 1$ servers. The i'th entry in the vector of signatures can be generated (and verified) with a key K_i that is shared by all elements of the i'th subset G_i of $2f + 1$ servers, $1 \leq i \leq \binom{n}{2f+1}$. For each K_i, the MAC scheme used to generate MAC values is common knowledge, but K_i is secret (unless divulged by some faulty server in G_i).

Sign The protocol to sign a message is shown in Figure 5.2. To sign a message, the signer sends a request to all the servers. A server generates the

```
Signature Client-Sign (Msg Msg) {
    ∀s : 1 ≤ s ≤ n, 1 ≤ g ≤ N, MACS[s][g] :=⊥
    ∀g : 1 ≤ g ≤ N, signature[g] :=⊥

    send ⟨SIGN, Msg, signer⟩ to all.

    // gather responses until we have f + 1 matching
    // MAC values for each of the N macs.
    while ( ∃g : signature[g] =⊥ )
    {
        on receive ⟨MACS, macs_i[1 . . . N]⟩ from server i.
            foreach g = 1 . . . N
                if (i ∈ G_g)
                    MACS[i][g] := macs_i[g]
                if ( ∃x ≠⊥: |{i : MACS[i][g] = x}| ≥ f + 1 )
                    signature[g] := x
    }
    return signature[1 . . . N];
}

void Signing-Witness-Server(Id i) {
    ∀g : 1 ≤ g ≤ N, macs_i[g] =⊥

    while(true) {
        rcv ⟨SIGN, Msg, signer⟩ from signer

        for each g in 1 . . . N
            if( i ∉ G_g )
                macs_i[g] :=⊥;
            else
                macs_i[g] := HMAC(signer : Msg, K_g);

        send ⟨MACS, macs_i[1 . . . N]⟩ to signer
    }
}
```

Figure 5.2: Signing procedures for MAC signatures.

MAC values for each group G_i that it belongs to and sends these values to the signer. The signer collects responses until it receives $(f + 1)$ identical MAC values for every group because each G_i contains at least $f + 1$ correct servers. Also, receiving $(f + 1)$ identical MAC values guarantees that the MAC value is correct because one of the values must be from a non-faulty server.

```
(bool, Signature) Client-Verify(Msg Msg, Signer S, Signature signature[1...N]) {
    {
        ∀s : 1 ≤ s ≤ n, received_responses[s] := false;
        ∀g : 1 ≤ g ≤ N, yes_count[g] := 0;
        ∀g : 1 ≤ g ≤ N, no_count[g] := 0;

        send ⟨VERIFY, Msg, signer, signature[1..N]⟩ to all.

        // gather responses until we have f + 1 matching
        // responses for each of the N MACs.
        while ( ∃g : yes_count[g] < f + 1 )
        {
            on receive ⟨RESULT, res_i[]⟩ from server i.
                foreach g := 1...N
                    if (i ∉ G_g)
                        continue;
                    else if (res_i[g] = CORRECT)
                        yes_count[g] = yes_count[g] + 1;
                    else if (res_i[g] = WRONG)
                        no_count[g] = no_count[g] + 1;

                    // If even one MAC-value is bad, we fail.
                    if (no_count[g] ≥ f + 1)
                        return (false, signature);
        }

        return (true, signature);
    }

void Verifying-Witness-Server(Id j) {
    ∀g : 1 ≤ g ≤ N, result_i[g] :=⊥

    while(true) {
        on receive ⟨VERIFY, Msg, signer, signature[1...N]⟩ from a verifier
            foreach g in 1...N
                if( i ∉ G_g )
                    result_i[g] :=⊥
                else if (signature[g] = HMAC(signer : Msg, K_g))
                    result_i[g] := CORRECT;
                else
                    result_i[g] := WRONG;

            send ⟨RESULT, result_i[1...N]⟩ to the verifier
    }
}
```

Figure 5.3: Verifying procedures for MAC signatures.

Verify The protocol to verify a signature is shown in Figure 5.3. To verify a signature, the verifier sends the signature, comprising of the N MACs to all the servers, along with the message. The i'th entry M_i is verified by server

p if $p \in G_i$ and M_i is the correct MAC value generated using K_i. A verifier accepts the signature if each entry in the vector is correctly verified by $f + 1$ servers. The verifier rejects a signature if at least one of its entries is rejected by $f + 1$ servers. Since the underlying MAC schemes are deterministic and each G_i contains $2f + 1$ servers, a signature is accepted by one correct verifier if an only if it is accepted by every other correct verifier and no other signature is accepted for a given message.

Correctness We now show that the Sign and Verify algorithms presented in Figure 5.2 and 5.3 implement a deterministic digital signature scheme.

Lemma 1. A signature consisting of correct MAC values at all the N positions will pass the verification procedure.

Proof. A non-faulty server will never send WRONG if the MAC value is correct. Thus, the verification procedure can get no more than f WRONG responses (from the faulty servers) for any of the N MACs in the vector.

Each MAC in the vector can be verified by $(2f + 1)$ servers, at least $(f + 1)$ of which are non-faulty. These non-faulty servers will send a response that the MAC value is CORRECT. Thus, on receiving the responses from all non-faulty servers, the verification procedure will accept the signature as it will have $(f + 1)$ CORRECT responses for each of the N positions in the vector. □

Lemma 2 (DS-Consistency). The signature generated by the signature procedure in Figure 5.2 contains the correct MAC values for every position in the vector.

Proof. The signer waits for $(f+1)$ matching MAC values for each of the N positions in the vector of MACs. Since at least one of the $(f+1)$ responses is from a non-faulty server, each of the MAC in the vector is correct. □

Lemma 3 (DS-Validity). A vector containing all N correct MACs cannot be generated unless the signer invokes the signing procedure.

Proof. For any set of f faulty servers, there is a group of $(2f+1)$ nodes that contains only non-faulty servers in it (because $n \geq 3f+1$). The MAC value corresponding to this group can only be generated correctly by one of these non-faulty servers, which will only do so if they receive a SIGN message from the signer. □

Lemma 4 (DS-Verifiability). If a signature passes the verification procedure in Figure 5.3, then the signature contains the correct MAC value for every position in the vector.

Proof. The verification procedure waits for $(f+1)$ CORRECT responses for each of the N MACs in the vector. Since at least one of the $(f+1)$ responses is from a non-faulty server, and a non-faulty server sends a CORRECT response only if the MAC is correct, each of the N MACs in the vector is correct. □

Theorem 8. The algorithm presented in Figures 5.2 and 5.3 implements a deterministic digital signature scheme.

Comments The Sign and Verify methods in Figure 5.2 and 5.3 satisfy all the properties of deterministic digital signatures. Thus, they can be used to implement the DS-Sign and DS-Verify functionality in Figure 5.1 to implement ∞-verifiable channels.

Unfortunately, the number of MACs required to implement the signature functionality grows exponentially with the number of servers. We show that one cannot do better and that trying to implement a deterministic digital signature scheme using MACs in this model *requires* an exponential number of MACs.

5.4.3 Complexity of deterministic digital signature implementations

We consider a general implementation that uses M secret keys. Every key K_i is shared by a subset of the servers; this is the set of servers that can generate and verify MAC values using K_i. We do not make any assumptions on how a signature looks. We simply assume that the signing procedure can be expressed as a deterministic function $\mathcal{S}(msg, k_1, k_2, \ldots, k_M)$ of the message to be signed (msg), where $k_1, \ldots, k_M$ are the values of the keys $K_1, \ldots, K_M$ used in the underlying MAC schemes.

The lower bound proof relies on two main lemmas which establish that (1) every key value must be known by at least $2f + 1$ servers, and (2) for any set of f servers, there must exist a key value that is not known by any element of the set. Then, we use a combinatorial argument to derive a lower bound on the number of keys.

Since we are proving a lower bound on the number of keys, we assume that the signature scheme uses the minimum possible number of keys. It follows, as shown in the following lemma, that no key is redundant. That is, for every key K_i, the value of the signature depends on the value of K_i for some message and for some combination of the values of the other keys.

Lemma 5 (**No key is redundant**). For every key K_i used in the deterministic

digital signature scheme $\mathcal{S}$, there are two different key values k_i^α and k_i^β and a message for which $\mathcal{S}$ produces different signatures when $K_i = k_i^\alpha$ and $K_i = k_i^\beta$.

$$\forall i, \exists msg, k_1, k_2, \ldots k_{i-1}, k_i^\alpha, k_i^\beta, k_{i+1}, k_{i+2}, \ldots, k_M :$$
$$\mathcal{S}(msg, k_1, k_2, \ldots, k_{i-1}, k_i^\alpha, k_{i+1}, \ldots, k_M) = \sigma_1,$$
$$\mathcal{S}(msg, k_1, k_2, \ldots, k_{i-1}, k_i^\beta, k_{i+1}, \ldots, k_M) = \sigma_2,$$
$$\text{and } \sigma_1 \neq \sigma_2$$

Proof. If the signature produced for a message is always independent of the key K_i, we can get an equivalent signature implementation by using a constant value for K_i. The resulting signature implementation will have one fewer key (since it does not use K_i), contradicting the assumption that the signature scheme uses the minimum possible number of keys. □

Now, we can establish the two main lemmas.

Lemma 6 ($2f+1$ **servers know every key)**. At least $(2f+1)$ servers know the value of K_i.

Proof. Proof by contradiction. Assume that key K_i is only known by a group G of servers, where $|G| \leq 2f$. Since $|G| \leq 2f$, G is the union of two disjoint sets A and B of size at most f each. From Lemma 5,

$$\forall i, \exists msg, k_1, k_2, \ldots k_{i-1}, k_i^\alpha, k_i^\beta, k_{i+1}, k_{i+2}, \ldots, k_M :$$
$$\mathcal{S}(msg, k_1, k_2, \ldots, k_{i-1}, k_i^\alpha, k_{i+1}, \ldots, k_M) = \sigma_1,$$
$$\mathcal{S}(msg, k_1, k_2, \ldots, k_{i-1}, k_i^\beta, k_{i+1}, \ldots, k_M) = \sigma_2,$$
$$\text{and } \sigma_1 \neq \sigma_2$$

Consider the following executions, where message msg is being signed. In all executions, the value of K_j is k_j for $j \neq i$.

- (Exec α) The symmetric key value for K_i is k_i^α. All servers behave correctly. The resulting signature value is σ_1.
- (Exec α') The symmetric key value for K_i is k_i^α. Servers not in B behave correctly. Servers in B set the value of K_i to be k_i^β instead of k_i^α. The resulting signature value is also σ_1 because the signature scheme is deterministic and tolerates up to f Byzantine failures and $|B| \leq f$.
- (Exec β) The symmetric key value for K_i is k_i^β. All servers behave correctly. The resulting signature value is σ_2.
- (Exec β') The symmetric key value for K_i is k_i^β. Servers not in A behave correctly. Servers in A set the value of K_i to be k_i^α instead of k_i^β. The resulting signature value is also σ_2 because the signature scheme is deterministic and tolerates up to f Byzantine failures and $|A| \leq f$.

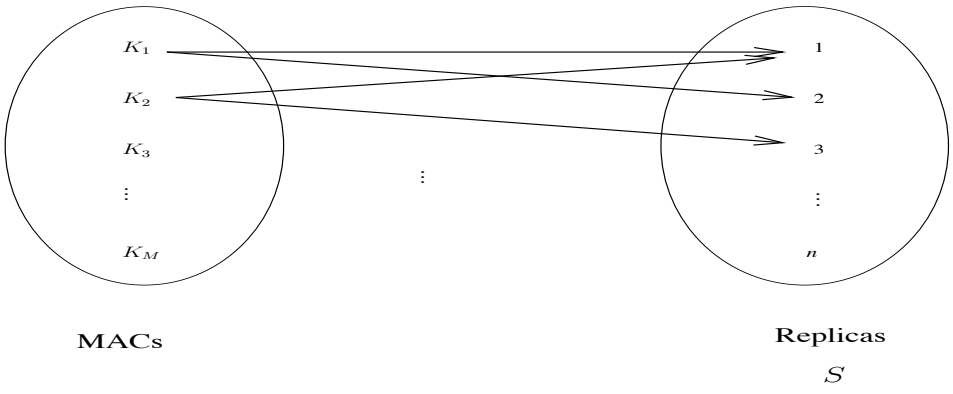
Executions α' and β' only differ in the identities of the faulty servers and are otherwise indistinguishable to servers not in G and to clients. Thus, the resulting signatures in both cases should be the same, implying $\sigma_1 = \sigma_2$, which is a contradiction. □

Lemma 7 (**Faulty servers do not know some key**). For every set of f servers, there exists a secret key K_i that no server in the set knows.

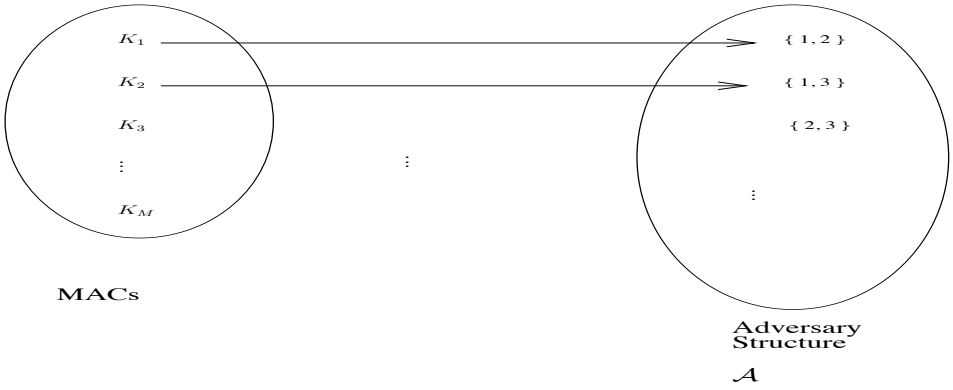
Proof. If a given set of f servers has access to all the M secret keys, then, if all the elements of the set are faulty, they can generate signatures for messages that were not signed by the signer, violating DS-Validity. □

We now establish a lower bound on the number of keys required by a MAC-based deterministic digital signature implementation.

Theorem 9. Any MAC-based implementation of a unique signature scheme requires at least $\binom{n}{f}/\binom{n-(2f+1)}{f}$ keys.



(a) Graph G



(b) Graph G'

Figure 5.4: Example bipartite graphs G and G'.

Proof. Let $G = (\mathcal{K}, R, E)$ be a bipartite graph over the set of keys $\mathcal{K} =$

$\{K_1, K_2, \ldots, K_M\}$ and the set of servers R, such that there is an edge between the key K_i and server j if and only if j is not assigned the key K_i by the unique signature scheme (that is j is not given the value of K_i as part of the scheme). Since every K_i is known by at least $(2f+1)$ servers (Lemma 6) it follows that, in G, the degree of vertex K_i is at most $\big(n-(2f+1)\big)$.

Let $G' = (\mathcal{K}, \mathcal{A}, E')$ be the bipartite graph over $\mathcal{K}$ and the set $\mathcal{A}$ of f-subsets of R. The set $\mathcal{A}$ consists of those subsets of R that could be controlled by the adversary. An edge (K, A) belongs to E' if and only if (K, a) belongs to E for every $a \in A$.

$$(K, A) \in E' \iff \forall a \in A : (K, a) \in E$$

If a node $K_i \in \mathcal{K}$ has degree d_i in G, then in G', K_i would have a degree $d'_i = \binom{d_i}{f} \leq \binom{n-(2f+1)}{f}$. From Lemma 7, it follows that for each $A \in \mathcal{A}$, $d'_A \geq 1$ in G'.

In any graph, the sum of out-degree must be equal to the sum of in-degree. From graph G', it follows that:

$$\begin{aligned} \sum_{k_i \in \mathcal{K}} d'_i &= \sum_{A \in \mathcal{A}} d'_A \\ \Rightarrow \quad \sum_{k_i \in \mathcal{K}} \binom{n-(2f+1)}{f} &\geq \sum_{A \in \mathcal{A}} 1 \\ \Rightarrow \quad M &\geq \binom{n}{f} \Big/ \binom{n-(2f+1)}{f} \end{aligned}$$

□

It follows that for $n = 3f+1$, the unique signature implementation described in Section 5.4.2 is optimal. In general, if the fraction of faulty nodes $\frac{f}{n} > \frac{1}{k}$, for $k \geq 3$, then the number of MACs required is at least $(\frac{k}{k-2})^f$.

5.5 Non-deterministic digital signatures

We now present a signature scheme, called matrix signatures, that does not require an exponential number of MAC implementations. Unlike the deterministic signature scheme presented in Section 5.4, the matrix signature scheme is non-deterministic. Thus, it is possible that there may be certain signatures that may non-deterministically be accepted or rejected by the verifier. The signature scheme provides DS-consistency and DS-validity, but not DS-verifiability. Matrix signatures provide a weaker set of properties, but one that is still sufficient to implement ∞-verifiable channels.

Matrix signatures are a non-deterministic signature scheme defined over a set of signers S and a set of verifiers V and consisting of a signing method MS-Sign$_{S,V}$ and a verification method MS-Verify$_{S,V}$:

$$\text{MS-Sign}_{S,V} \quad : \quad \Sigma^* \mapsto \Sigma^* \tag{5.5.1}$$

$$\text{MS-Verify}_{S,V} \quad : \quad \Sigma^* \times \Sigma^* \mapsto \textbf{Boolean} \times \Sigma^* \tag{5.5.2}$$

The signing procedure MS-Sign$_{S,V}$ is used to sign a message. It outputs a signature, which can convince the verifier that the message was signed. The set S contains all the processes that can invoke the signing procedure. The set V contains all processes that may verify a signature in the signature scheme. The verification procedure, MS-Verify$_{S,V}$, takes as input a message and a signature and outputs two values. The first value is a boolean and indicates whether the verification procedure *accepts* or *rejects* the signature. The second value returned by the verification method is a signature, whose role needs some explaining.

Matrix signature schemes guarantee that (i) a verifier always accepts signatures that are generated by invoking the signing procedure and that (ii)

any message whose signature is accepted was, at some point, signed by a member of S by invoking the signing procedure. A signature accepted by the verification procedure does not have to be the one produced by the signing procedure. We call these second type of signatures *derivative*. Derivative signatures present a challenge in terms of ensuring verifiability, as they can either be non-deterministically accepted or rejected by the verification procedure. To address this challenge, we require the verification procedure to produce as output a new derivative signature that – by construction – is guaranteed to be accepted by all verifiers whenever the original signature is accepted. This new signature can then be used by v to authenticate the sender of m to all other verifiers. If the first output value produced by the verification procedure is **false**, then the second output value is irrelevant.

Matrix signature schemes satisfy the following properties:

- (**MS-Consistency**) A signature produced by the signing procedure is accepted by the verification procedure.

$$\text{MS-Sign}_{S,V}(msg) = \sigma \Rightarrow \text{MS-Verify}_{S,V}(msg, \sigma) = (true, \sigma')$$

- (**MS-Validity**) A signature for a message m that is accepted by the verification procedure cannot be generated unless a member of S has invoked the signing procedure.

$$\text{MS-Verify}_{S,V}(msg, \sigma) = (true, \sigma') \Rightarrow \text{MS-Sign}_{S,V}(msg) \text{ was invoked}$$

- (**MS-Verifiability**) If a signature is accepted by the verification procedure for a message m, then the verifier can produce a signature for m that is guaranteed to be accepted by the verification procedure.

$$\text{MS-Sign}_{S,V}(msg, \sigma) = (true, \sigma') \Rightarrow \text{MS-Verify}_{S,V}(msg, \sigma') = (true, \sigma")$$

MS-Verifiability is recursively defined; it ensures verifiability. If the verification procedure accepts a signature for a given message, then it outputs a signature that is accepted by the verification procedure for the same message. In turn, the output signature can be used to obtain another signature that will be accepted by the verification procedure and so on.

```
VerifiableChannel extends Channel {
    Verifiable-Send(Message msg) {
        σ_msg = MS-Sign(K_s, msg);
        Channel-Send(⟨msg, σ_msg⟩);
    }

    // receiver r_1
    (Message, proof) Verifiable-Deliver() {
        do {
            ⟨msg, σ⟩ = Channel-Deliver();
            if (MS-Verify(K_v, msg, σ) = (true, σ'))
                return (msg, σ');
        } while (true);
    }

    // process r_i for i ≥ 1
    Boolean SentByTheSender( msg, proof ) {
        if (MS-Verify(K_v, msg, proof) = true)
            return true;
        return false
    }
}
```

Figure 5.5: Alternative implementation: Verifiable channel.

We now show that matrix signatures are sufficient to implement ∞-verifiable channels. Figure 5.5 shows an implementation of the ∞-verifiable channel using matrix signatures. The sender s signs every message msg that is sent on the channel $C(s, r_0)$ using the matrix signature scheme, where s is the only signer in the set S and the set V contains all the processes in the system that may receive the message. The sender sends the message along with the signature σ_{msg} to the receiver r_0. To deliver a message, receiver r_0 verifies the matrix signature, σ, for the message and delivers the message only if the

MS-Verify procedure returns **true**. This convinces the receiver that the sender must have sent the message. Now, any correct process r_i that has verified the signature can send the new signature σ', as a proof, to convince any other correct process r_{i+1} that the message was sent by the sender s.

Theorem 10. The algorithm in Figure 5.1 implements an ∞-verifiable channel.

Proof. A correct process r accepts that a message msg was sent by the sender s if and only if msg is accompanied by a valid signature σ_{msg} from s. Thus,

1. A correct receiver r_0 delivers a message msg only if accompanied by a valid digital signature σ_{msg} for the message. The signature σ_{msg} serves as a proof that will be accepted by any correct process r_1.

2. Any correct process r_i that accepts σ_{msg} as a proof will only do so after verifying that MS-Verify$(K_v, msg, \sigma_{msg}) = (\textbf{true}, \sigma'_{msg})$. MS-Verifiability guarantees that r_i can use σ'_{msg} as a proof $\pi_{i+1}(s, msg)$ that will be accepted by any correct process r_{i+1}.

3. From MS-Validity, it follows that a correct process r will not accept a proof $\pi(s, msg')$ for a message msg' that s has not sent.

□

5.5.1 Matrix signature implementation

We now show how matrix signatures can be implemented.

A matrix signature consists of n^2 MAC values arranged in n rows and n columns, which together captures the servers' collective knowledge about the authenticity of a message.

We have $n \geq 3f + 1$ witness servers which share pairwise secret keys that are used to generate/verify MACs. Each witness server functions both as a signing witness server to implement the signing witness, and as a verifying witness server to implement the verifying witness.

Clients can sign (or verify a signature) by contacting all the signing witness (or, respectively, verifying witness) servers. The key difference with the protocol described in the previous section is that the signature being used is a matrix of $n \times n$ MACs as opposed to a single MAC value. Each MAC value in the matrix is calculated using a secret key $K_{i,j}$ shared between a signing witness server i and a verifying witness server j.[1]

The i^{th} row of the matrix signature consists of the MACs generated *by* the i^{th} signing witness server. The j^{th} column of the matrix signature consists of the MACs generated *for* the j^{th} verifying witness server. In Figure 5.6, the row in bold font is generated by the 2^{nd} signing witness server, and the column in bold is generated for the 3^{rd} verifying witness server.

We distinguish between *valid* and *admissible* matrix signatures:

Definition (Valid). A matrix signature is *valid* if it has at least $(f + 1)$ correct MAC values in every column.

Definition (Admissible). A matrix signature is said to be *admissible* if it has at least one column corresponding to a non-faulty server that contains at least $(f + 1)$ correct MAC values.

An *admissible* matrix signature captures the minimum requirement for it to be successfully verified by a non-faulty verifier. A *valid* matrix signa-

[1] Although a signing witness server and a verifying witness server can both be mapped to a single witness server, for the time being it is useful to think of them as separate entities.

$h_{1,1}$	$h_{1,2}$	$\mathbf{h_{1,3}}$	$h_{1,4}$
$\mathbf{h_{2,1}}$	$\mathbf{h_{2,2}}$	$\mathbf{h_{2,3}}$	$\mathbf{h_{2,4}}$
$h_{3,1}$	$h_{3,2}$	$\mathbf{h_{3,3}}$	$h_{3,4}$
$h_{4,1}$	$h_{4,2}$	$\mathbf{h_{4,3}}$	$h_{4,4}$

(a) A Matrix-signature

$h_{1,1}$	$h_{1,2}$	$h_{1,3}$	$h_{1,4}$
$h_{2,1}$	$h_{2,2}$	$h_{2,3}$	$h_{2,4}$
?	?	?	?
?	?	?	?

(b) A Valid Signature

?	$h_{1,2}$	?	?
?	$h_{2,2}$	?	?
?	?	?	?
?	?	?	?

(c) An Admissible Signature

Figure 5.6: Example matrix-signatures.

ture captures the minimum requirement for being *guaranteed* to be always successfully verified by any non-faulty verifier. Thus, every *valid* signature is *admissible*, but the converse does not hold.

The protocol for generating and verifying matrix signatures is shown in Figures 5.7 and 5.8.

Generating a Signature To generate a matrix signature, the signer s sends the message *msg* to be signed, along with its identity, to all the signing witness servers over *authenticated channels*. Each signing witness server generates a

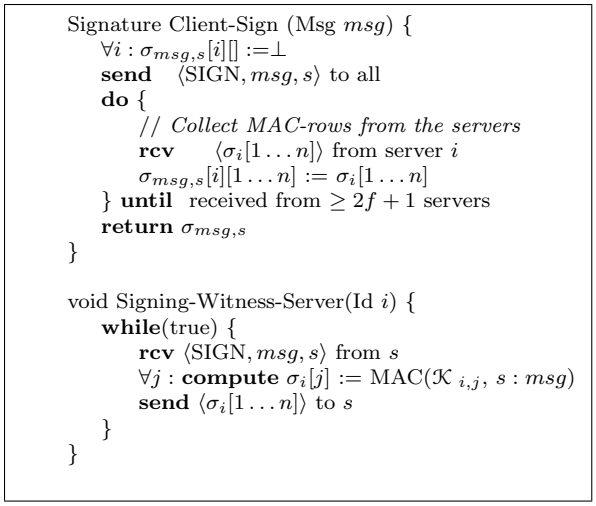


Figure 5.7: Generating matrix signatures.

row of MACs, attesting that s signs msg, and responds to the signer. The signer waits to collect the MAC rows from at least $(2f + 1)$ signing witness servers to form the matrix signature.

The matrix signature may contain some empty rows corresponding to the unresponsive/slow servers. It may also contain up to f rows with incorrect MAC values, corresponding to the faulty servers.

Verifying a Signature To verify a matrix signature the verifier sends to the verifying witness servers: (a) the matrix signature, (b) the message, and (c) the identity of the client claiming to be the signer. A verifying witness server admits the matrix signature only if at least $(f + 1)$ MAC values in the server's column are correct; otherwise, it rejects the matrix signature. Note that a non-faulty server will never reject a valid matrix signature.

The verifier collects responses from the servers until it either receives $(2f+1)$ $\langle \text{ADMIT}, \ldots \rangle$ responses to accept the matrix signature, or it receives

```
(bool, Signature) Client-Verify(Msg msg, Signer s, Signature σ) {
    ∀i : σ_new[i][] :=⊥
    ∀i : resp[i] :=⊥
    send  ⟨VERIFY, msg, s, σ[][]⟩ to all
    do {
        rcv ⟨ADMIT, σ_i[1...n]⟩ or ⟨REJECT⟩ from server i
        if received ⟨ADMIT, σ_i[1...n]⟩ {
            σ_new[i][1...n] := σ_i[1...n]
            resp[i] := ADMIT
            // Accept: If (2f + 1) admit it
            if ( Count(resp, ADMIT) ≥ 2f + 1 )
                return (true, σ_new);
        } else {
            if (resp[i] =⊥)
                resp[i] := REJECT
            // Reject: If at least one non-faulty server rejects
            if ( Count(resp, REJECT) ≥ f + 1 )
                return (false, ⊥);
        }

        // If cannot accept or reject: retry with σ_new
        if ( Count(resp, ADMIT) + Count(resp, REJECT) ≥ (n − f) ) {
            send  ⟨VERIFY, msg, s, σ_new[][]⟩ to { r : resp[r] ≠ ADMIT}
        }
    } until (false)
}

void Verifying-Witness-Server(Id j) {
    while(true) {
        rcv ⟨VERIFY, msg, s, σ⟩ from V
        correct_cnt := |{i : σ[i][j] == MAC(K_{i,j}, s : msg)}|
        if (correct_cnt ≥ f + 1)
            ∀l : compute σ_j[l] := MAC(K_{j,l}, s : msg)
            send ⟨ADMIT, σ_j[1...n]⟩ to V
        else
            send ⟨REJECT⟩ to V
    }
}
```

Figure 5.8: Verifying matrix signatures.

$(f+1)$ $\langle$REJECT$\rangle$ responses to reject the matrix signature as not *valid*.

Regenerating a valid matrix signature Receiving $(2f+1)$ $\langle$ADMIT, ...$\rangle$ responses does not guarantee that the matrix signature being verified is *valid*. If some of these responses are from Byzantine nodes, the same matrix signature could later fail the verification if the Byzantine nodes respond differently.

Verifiability requires that that if a matrix signature passes the verification procedure, then the verifier gets a matrix signature that will always pass the verification procedure. This is accomplished by constructing a new matrix signature that is a *valid* matrix signature.

Each witness-server acts both as a verifying witness server and a signing witness server. Thus, when a witness-server admits a matrix signature (as a verifying witness server), it also re-generates the corresponding row of MAC values (as a signing witness server) and includes that in the response. Thus, if a verifier collects $(2f+1)$ $\langle \text{ADMIT}, \ldots \rangle$ responses, it receives $(2f+1)$ rows of MAC values, which forms a *valid* matrix signature.

Ensuring termination The verifier may receive $(n-f)$ responses, and still not have enough admit responses or enough reject responses, to decide. This can happen if the matrix signature being verified, σ, is maliciously constructed such that some of the columns are bad. This can also happen if the matrix signature σ is *valid*, but some non-faulty servers are slow and Byzantine servers, who respond faster, reject it.

To ensure receipt of $(2f+1)$ $\langle \text{ADMIT}, \ldots \rangle$ responses, the verifier retries by sending σ_{new}, each time σ_{new} is updated, to all the servers that have not sent an $\langle \text{ADMIT}, \ldots \rangle$ response. Eventually, the verifier either receives $(f+1)$ $\langle \text{REJECT} \rangle$ responses from different servers (which guarantees that σ was not *valid*), or it receives $(2f+1)$ $\langle \text{ADMIT}, \ldots \rangle$ responses (which ensures that the regenerated matrix signature, σ_{new} is *valid*).

5.5.2 Correctness

We now show that if $n \geq 3f + 1$, then matrix-signatures satisfy all the requirements of digital signatures and ensure that the signing/verification procedures always terminate.

Lemma 8. Every valid matrix signature is admissible.

Proof. A valid matrix signature by definition has at least $(f + 1)$ rows of all correct MAC values. Every column has at least $(f+1)$ correct MAC values. □

Lemma 9. The matrix signatures generated by a non-faulty signer (Figure 5.7) is valid.

Proof. A non-faulty signer has to collect MAC rows from at least $(2f + 1)$ servers to generate a matrix signature. At least $(f + 1)$ of these rows are from non-faulty servers and consist only of correct MAC values. □

Lemma 10. A valid matrix signature always passes the verification procedure for a non-faulty verifier.

Proof. A valid matrix signature consists of all correct MAC values in at least $(f + 1)$ rows. So, no non-faulty server will send a $\langle$REJECT$\rangle$ message. When all non-faulty servers respond, the verifier will have $(2f + 1)$ $\langle$ADMIT, ...$\rangle$ messages. □

Lemma 11. If a matrix signature is not-admissible, then it will fail the verification procedure for any non-faulty verifier.

Proof. If a matrix signature is not admissible, then all non-faulty servers will reject it by sending the $\langle$REJECT$\rangle$ message. On receiving $(n - f)$ responses,

the verifier will have $(n - 2f) \geq (f + 1)$ $\langle$REJECT$\rangle$ messages causing the verification procedure to fail. □

Lemma 12. If a matrix signature passes the verification procedure for a non-faulty verifier, then it is admissible. □

Lemma 13. An adversary cannot generate an admissible matrix signature for a message msg, for which the signer did not initiate the signing procedure.

Proof. Consider the first admissible matrix signature, σ, that is generated for msg. By the definition, σ should have at least $(f + 1)$ correct MAC values in a column corresponding to a non-faulty server (say j). At least one of these MAC values is in a row that corresponds to a non-faulty server (say i). The key $K_{i,j}$ is only known to the non-faulty servers i and j.

Only server i might generate the MAC value and give it to a client. If the message was not signed by the signer, then the MAC must have been generated as part of the verification procedure. In the verification procedure, a non-faulty server i only generates the MAC value if it has already received an admissible matrix signature that has $(f + 1)$ correct MAC values in column i. This is not possible because σ is the first admissible matrix signature generated. □

Lemma 14. If a matrix signature passes the verification procedure for a non-faulty verifier, then the newly reconstructed matrix signature is valid.

Proof. For a matrix signature to pass the verification procedure, the verifier must receive at least $(2f + 1)$ $\langle$ADMIT, ...$\rangle$ responses. At least $(f + 1)$ of these are from non-faulty servers and include a correct MAC row along with

the response. Thus the reconstructed matrix signature consists of at least $(f + 1)$ correct rows. □

Lemma 15. If a non-faulty verifier accepts that s has signed msg, then it can convince every other non-faulty verifier that s has signed msg.

Proof. A non-faulty verifier, v_1, accepts that a message is signed only if it passes the verification procedure. The newly generated matrix signature that it gathers, σ_{new}, is *valid* and will pass the verification for any non-faulty verifier v_2, convincing the verifier (v_2) that the message was signed. □

Theorem 11. The matrix-signature scheme presented in Figures 5.7 and 5.8 satisfies *consistency*, *validity* and *verifiability*.

Proof. Consistency follows from Lemmas 9 and 10. Validity follows from Lemmas 12 and 13. Verifiability follows from Lemmas 10 and 14. □

Theorem 12. If $n \geq 3f + 1$ the signing procedure always terminates for any non-faulty signer.

Proof. There are at least $(n - f)$ non-faulty servers that will respond to the signer. Thus eventually, it will get $(n - f) \geq (2f + 1)$ responses. □

Theorem 13. If $n \geq 3f + 1$ the verification procedure always terminates for any non-faulty verifier.

Proof. Suppose that the verifier does not terminate even when it gets the responses from all the non-faulty servers. It cannot have received more than f $\langle$REJECT$\rangle$ responses. Thus, it would have received at least $(f + 1)$ $\langle$ADMIT, ...$\rangle$ responses from the non-faulty servers that is accompanied with

the correct row of MACs. These $(f+1)$ rows of correct MACs will ensure that the new matrix signature σ_{new} is valid.

Thus all non-faulty servers that have not sent a $\langle \text{ADMIT}, \ldots \rangle$ response will do so when the verifier retries with σ_{new}. The verifier will eventually have $(n - f) \geq (2f + 1)$ $\langle \text{ADMIT}, \ldots \rangle$ responses thus enabling the verification procedure to terminate. □

Discussion Many practical Byzantine fault tolerant (BFT) state machine replication [34, 37, 48, 80] have used MAC-based verifiable channels to improve performance. However, such constructions could only provide verifiability up to two hops. It has been shown that relying on 2-verifiable channels instead of ∞-verifiable channels exposes these system to advanced attacks in which the system makes little progress [45]. The UpRight system uses MAC-based implementation of the ∞-verifiable channels to prevent these attacks [44].

Chapter 6

The authorized channel

In this chapter we discuss the authorized channel and its implementations. First, we revisit the definition in Section 6.1. In Section 6.2, we present an implementation based on digital signatures. Later, in Section 6.3 we present implementations based on secret sharing.

6.1 The authorized channel

A distributed protocol specifies the messages that each participating process is allowed to send, depending upon the process' initial state and the sequence of messages it has delivered. While correct processes follow the protocol exactly, faulty processes may skip sending or receiving the appropriate message (omission failures) or may send messages that they are not supposed to send (Byzantine failures).

An authorized channel aims to restrict the harm that a malicious sender can inflict on a correct receiver. Authorized channels ensure that a correct receiver will only deliver messages that the sender is authorized to send. Thus, any unauthorized message that a malicious sender may attempt to send will not be delivered by the receiver.

For example, many atomic storage protocols require a reader to write-back the value it read, to ensure that a later read operation does not return

an older value. If readers are trusted to follow the protocol, then the servers may accept a write-back from the reader without any checks. However, if readers can be faulty, malicious readers may spoil the state of the system by writing-back a value that has never been written by the writer. Accepting such a write-back message, without any checks, can violate the system's safety requirements.

It is thus important to ensure that certain messages be accepted only after validating that the sender is authorized to send the message. An authorized channel, C_{auth}, formalizes such a requirement by ensuring that a (correct) receiver delivers a message only if the sender (who may or may not be correct) is authorized to send the message. For convenience, we repeat the definition given in Chapter 2:

Definition. Let $P(s, msg, \tau)$ be a predicate that evaluates to **true** only if the sender s is authorized to send the message msg by global-time τ. An authorized channel guarantees that a correct receiver r shall deliver a message msg from the channel $C(s, r)$, only if the sender is authorized to send the message.

$$\text{Authorized channel} : msg \in Delivered_{r,C}(\tau) \Rightarrow P(s, msg, \tau)$$

6.2 Implementing authorized channels

If the sender process is known to be benign, then authorized channels can be trivially implemented without requiring any action on the part of the receiver. However, if the sender may be malicious, then the receiver must verify that the sender is authorized to send a message msg before delivering it. To enable the receiver to verify that the sender is authorized to send the message, the sender s will have to provide a proof for the same.

In general, verifying that the sender is authorized to send a message involves validating (a) the initial state of the sender, (b) the local history (of local or send/receive events) at the sender, and (c) the protocol to be followed by the sender [19, 28, 46]. Validating all these items is not always possible. First, in many protocols, a malicious process may lie about its initial state by reporting a state σ' instead of σ without being detected. Second, in asynchronous systems, a process may lie about the order in which it received messages. Third, for non-deterministic protocols, a malicious process can lie about the outcomes of non-deterministic events (such as coin-tosses etc.) that are locally generated. In such cases, we can only ensure that a malicious process behaves consistently with other processes [19].

Fortunately, for many distributed systems the set of conditions that determine whether or not a process is authorized to send a message is a lot simpler. Many storage protocols require that processes initialize their local state to a common value that is known to all. In such cases, a malicious sender cannot lie about its initial state without being detected as faulty. Also, the precise order in which messages are received does not always matter. For example, in the case of an atomic storage protocol, the reader has to wait until it collects at least $(f + 1)$ identical responses before it performs a write-back. The order of message receipt does not effect whether or not the reader is authorized to perform a write-back. Finally, there are many protocols that are inherently deterministic.

6.2.1 Threshold authorized channels

Many distributed protocols replicate processes to achieve fault tolerance and improve performance. Such replicated state machine architectures

typically ensure that each replica sends the same the message to all other replicas and that replicas act on a received message only after receiving a threshold number of them from different replicas. For example, in many Byzantine fault tolerant systems a correct process waits for at least $f + 1$ identical responses to act on a received message. Threshold-authorized channels formalize this requirement.

Definition. Let $Next(msg_1, msg_2)$ be a predicate, denoting that message msg_2 is a valid response to msg_1; and let P be a set of processes. The authorizing predicate for a threshold-authorized channel $Auth(s, msg_{next}, H)$ allows a sender to send a message msg_{next} only if, given the current history H, the sender has received at least t messages $msg_1, msg_2, \ldots msg_t$, from different processes in P, such that $Next(msg_i, msg_{next})$ is **true**.

Threshold authorized channel : $msg \in Delivered_{r,C}(H) \Rightarrow Auth(s, msg, H)$

Current implementations of the threshold-authorized channels require that, in order to be authorized to send some message B, the sender s receive t instances of some "enabling message" A from t distinct processes. Each of these messages must be received by s over a verifiable channel: this will allow s to collect t proofs from distinct processes which together can convince any correct process that s is indeed authorized to send message B. Before delivering B, te intended recipient first verifies the proof that at least t different processes did send message A to s, and then delivers the message only if the proof is correct.

6.3 Alternative implementation using secret sharing

Although verifiable channels are sufficient to implement a threshold authorized channel, they are not necessary. Using verifiable channels to implement an authorized channel provides additional features that, while not required by authorized channels, come at a cost. For example, an authorized channel only needs to verify whether or not the sender has received at least t A messages, but does not need to distinguish between the case in which the sender has received exactly t messages from the one in which it has received $t + 1$. Similarly, the implementation of the authorized channel need not differentiate between the case where the sender has received $t - 1$ messages and the case where the sender has received $t - 2$ messages.

We show that threshold-authorized channels can be alternatively implemented using secret sharing techniques, which are not only result in a cheaper implementation but also provide superior, information-theoretic security guarantees.

6.3.1 Information theoretically secure channel

A trusted process $p_{trusted}$ is used to generate a random secret S_B for each message B that the sender might send.

1. $p_{trusted}$ splits the secret S_B into n shares such that t of these shares are necessary and sufficient to regenerate the secret.

2. The trusted process (i) sends the secret S_B to the receiver r, and (ii) distributes one share of the secret sh_i, over private channels, to each process $p_i, 1 \leq i \leq n$ that might send a message A_i.

3. Process p_i keeps sh_i a secret until it sends the message A_i to the sender s. On sending the message A_i to s, process p_i also reveals the secret share sh_i to the sender s.

   On receiving k messages from different processes, the sender s will have k such shares to reconstruct the secret.

4. To send message B on the channel C_{auth}, the sender s has to reconstruct the secret S'_B and provide it as a proof that s is authorized to send the message.

   Secret sharing techniques ensure that the sender will be able to reconstruct the secret correctly if and only if it has received message A from at least t different processes.

5. Receiver r delivers the message B only if the secret S'_B reconstructed by the sender is same as the secret S_B given by the trusted process.

Theorem 14. The protocol described in Section 6.3.1 implements a threshold-authorized channel.

Proof. The receiver r accepts the message B only if the sender s is able to reconstruct the secret correctly. The adversary cannot find out the secret as the shares are exchanged between correct processes over private channels. Thus, the sender can reconstruct the correct secret only if at least t processes have revealed their shares. Process p_i reveals its share only when it sends the appropriate message A_i. Hence, s can reconstruct the secret and convince r to accept B only if s receives at least t messages with their corresponding shares.

6.3.2 Computationally secure channel

The above implementation requires the use of private channels but it is secure even against a computationally unbounded adversary. Thus the adversary, no matter how powerful, cannot fool a correct receiver into accepting an unauthorized message from the sender with any probability higher than that of taking a random guess at the generated secret. If the generated secret is of length k_s bits, then the probability that the adversary guesses the secret correctly is $1/2^{k_s}$, which can be made as small as required by increasing the length of the generated secret. To achieve this information-theoretic security, however, the private channels used in the implementation need to be information-theoretically secure themselves.

Many practical systems however do not require information-theoretic security, either because they are willing to assume that the adversary is computationally bounded, or because certain other components of the system already require the adversary to be computationally bounded. For such situations, we can develop an alternative implementation of the authorized channel that uses secret sharing combined with one-way hashing; this implementation is secure as long as the computationally-bounded adversary is unable to reverse the one-way hash. The implementation is as follows:

A trusted process $p_{trusted}$ generates a random secret S_B for each message B that the sender might send.

1. $p_{trusted}$ splits the secret S_B into n shares such that t of these shares are necessary and sufficient to regenerate the secret.

2. The trusted process sends (i) the hash of the secret $h(S_B)$ to the receiver r, and (ii) privately distributes one share of the secret sh_i to each process

$p_i, 1 \leq 1 \leq n$ that might send a message A_i.

3. Process p_i keeps sh_i a secret until it sends the message A_i to the sender s. On sending the message A_i to s, process p_i also reveals the secret share sh_i to the sender s.

   On receiving k messages from different processes, the sender s will have k of the shares needed to reconstruct the secret.

4. To send message B on the channel C_{auth}, the sender s has to reconstruct the secret S'_B and provide it as a proof that s is authorized to send the message.

   Secret sharing techniques ensure that the sender will be able to reconstruct the secret correctly if and only if it has received message A from at least t different processes.

5. Receiver r delivers the message B only if the secret S'_B reconstructed by the sender produces the same hash as $h(S_B)$.

Theorem 15. The protocol described in Section 6.3.2 implements a threshold-authorized channel.

Proof. The receiver r accepts the message B only if the sender is able to reconstruct the secret correctly. A computationally-bounded adversary cannot invert the hash function, or access the shares that are exchanged between correct processes over private channels. Thus, the sender can reconstruct the correct secret only if at least t processes have revealed their shares. Process p_i reveals its share only when it sends the appropriate message A_i.

Discussion We make two observations. First, an authorized channel provides guarantees about the message being authorized only as long as the receiver is correct. Second, if the receiver r can communicate with all processes that send the message to authorize the sender, then the role of the trusted process can be fulfilled by the receiver r itself.

Related work Notions similar to that of an authorized channel have been used earlier, in the context of broadcast channels, to translate automatically crash-tolerant protocols into Byzantine fault tolerant systems [16, 19, 20, 28, 47, 46, 68, 69, 101, 102]. Bracha [28, 29] and Coan [46] have proposed such translation mechanisms for asynchronous systems. For synchronous systems, such mechanisms have been proposed by Bazzi, Neiger, Toueg and Mpoeleng [16, 19, 20, 99, 101, 102]. Recently, PeerReview [71] and Nysiad [77] have built upon such techniques to detect and mitigate Byzantine faults in scalable distributed systems [76].

Chapter 7

Background: Register abstractions

Register abstractions can be implemented using quorum systems [59, 118]. In this chapter we discuss existing quorum techniques to implement register abstractions and their trade-offs. Section 7.2 introduces strict quorum systems that are useful to implement registers with strong consistency semantics. Section 7.3 discusses non-strict quorum systems that are useful to implement registers that provide weaker consistency semantics in order to achieve better availability.

7.1 Preliminary definitions

Definition. A quorum system over a set of elements $\mathcal{P}$ is a tuple $\big(\mathcal{R}, \mathcal{W}\big)$; where $\mathcal{R} \subset 2^{\mathcal{P}}$ is the set of read quorums and $\mathcal{W} \subset 2^{\mathcal{P}}$ is the set of write quorums.

Quorum systems are widely used to implement distributed registers [15, 17, 52, 59, 61, 65, 89, 90, 91, 95, 94, 106, 118]. To implement a distributed register, we define a quorum system over the set of servers $\mathcal{P}$ that store the value. To perform a read operation on the register, the reader contacts a quorum of servers, R, from the set of read quorums $\mathcal{R}$, and chooses the latest value among the received values. Similarly, to perform a write operation, the writer contacts a quorum of servers, W, from the set of write quorums $\mathcal{W}$ and updates the value at the servers in W.

In order to ensure that a reader is able to read the value written by a writer, read quorums must intersect with write quorums. The requirement on the intersection depends on (i) the type of faults that a server may be subject to, and (ii) on the consistency semantics of the register that is implemented. We classify quorum systems into two types: strict quorum systems and non-strict quorum systems.

7.2 Strict quorum systems

Strict quorum systems require that any read and write quorum always intersect with each other. The number of servers required to be in the intersection depends on the kind of faults that the servers may be subject to:

1. If all the servers are correct, then the intersection needs to contain at least one server.

2. If the servers may be subject to benign failures (crash, omission etc.), then the intersection must be larger. To tolerate f benign server failures, the intersection needs to be at least $f + 1$. This ensures that every read quorum intersects with a write quorum in at least one correct server.

3. Finally, if the servers may be Byzantine, then the intersection may need to be even larger. This is to ensure that a reader is able to correctly identify a value returned by the correct servers:

    - If the data being written to the register is self-verifying in nature, i.e. modifications to the data written by the writer is detectable, then the intersection between the read and write quorums need to contain at least $f+1$ servers in order to tolerate f Byzantine faults.

- If the data being written is not self-verifying, then the intersection between the read and write quorums needs to contain at least $2f+1$ servers in order to tolerate f Byzantine servers [90].

Implementing registers: Because of their strong intersection property, strict quorum systems can be used to build registers that provide Lamport's *safe*, *regular*, and *atomic* semantics [85] and therefore guarantee that read operations that are not concurrent with a write operation return the result of the latest completed write. Unfortunately, these registers cannot provide good availability if there are many failures, or if there are any network partitions [57, 60].

7.3 Non-strict quorum systems

Non-strict quorum systems are designed to achieve higher availability at the cost of weakening the consistency guarantees. This is achieved by relaxing the intersection requirement found in strict quorum systems. Non-strict quorum systems studied in the literature are of two kinds: probabilistic quorum systems (PQS) and signed quorum systems (SQS).

Probabilistic quorum systems Probabilistic quorum systems consist of read and write quorums, along with an access strategy for choosing a quorum [92]. Two quorums chosen according to the specified access strategy are required to satisfy the intersection requirement with high probability; such that, for a given ϵ where $0 < \epsilon < 1$, the probability of satisfying the intersection requirement is at least $(1-\epsilon)$. As with the case of strict quorum systems, the requirements on the intersection is dependent on the kind of server failures

that can occur. Malkhi et. al. [92] present various intersection requirements depending on the kind of server failures tolerated and the nature of the data stored in the register.

Signed quorum systems Signed quorum systems [121] also provide a probabilistic guarantee of intersection between two quorums. Unlike probabilistic quorum systems, signed quorum systems use the notion of a failure detector [4, 38, 39, 41] to estimate which servers are responsive and which servers are not. Responsive servers are represented using positive elements, and non-responsive servers are represented using negative elements. A quorum may consist of both positive and negative elements, corresponding to the case where a server that is contacted during an operation either is responsive or is believed to be unresponsive. The intersection property requires that two quorum sets (i) either intersect in a positive element, i.e. a responsive server, or (ii) contain a minimum number of mismatches, where mismatch is the event where a server is considered to be responsive during one quorum access, but not during the other. If mismatches on different servers are independent, then signed quorum systems guarantee that either two quorums intersect in a positive element, or the probability that both quorums can be acquired is very small.

In a system with perfect failure-detectors, if the configuration of the nodes does not change significantly, then signed quorum systems will behave like a strict quorum system and provide safe semantics. However, with imperfect failure detectors, it is possible that the read and write quorums used do not intersect.

Implementing registers: If the network is synchronous, P-randomized register can be implemented using either probabilistic quorum systems or

signed quorum systems. However, if the network is asynchronous, an adversary may delay messages from the clients to various servers such that the probability of non-intersection cannot be bounded. This may result in the read operation returning arbitrarily old values or values that were never written to the system (in case of Byzantine faults).

7.3.1 Limitations of existing approaches

Quorum systems are commonly used to improve the availability and fault tolerance of a distributed service. However, asynchrony poses an interesting challenge to the implementation of such services. If the network is asynchronous, then it is not possible to distinguish effectively a crashed process from a process that is slow [56]. Thus, if a distributed storage system is prone to operating in conditions where the network can be asynchronous, or where there is a possibility of a network partition, then it is not possible to ensure both strong consistency and high availability [57, 60].

Strict quorum systems and non-strict quorum systems make different trade-offs in terms of handling such asynchronous conditions. Register implementations based on strict quorum systems provide strong consistency guarantees at the cost of being unavailable under adversarial conditions; implementations based on non-strict quorum systems allow for high availability by weakening the consistency guarantees they provide.

Unfortunately, with probabilistic quorum systems and signed quorum systems the probability that the read and write quorum sets do not intersect can be bound only when the network is synchronous. If the network is asynchronous, then neither quorum system can be used to implement a P-randomized register. Under asynchronous conditions, the adversary can

orchestrate the message delays from clients to different servers such that the read and write quorums never intersect. Thus, implementations based on these quorum systems may return arbitrarily old values. Moreover, if the systems is prone to Byzantine failures, then it is possible that the system may return maliciously-fabricated values that were never written to the system.

7.3.2 Alternative register abstractions

To address these concerns, Bazzi [5] proposes an alternative way to relax the consistency guarantees that allows for higher availability, while ensuring that the worst-case staleness is bounded even under adversarial conditions. In particular, the proposed k-safe, k-regular and k-atomic registers relax the consistency guarantees defined by Lamport [85] to allow the system to return any one of the latest k written values [5]. Unlike the P-random register, these new register abstractions do not focus on bounding the probability of returning stale values, but focus instead on bounding the worst possible staleness.

1. **k-safe(k):** A read that does not overlap with a write returns the result of one of the latest k completed writes. The result of a read overlapping a write is unspecified.

2. **k-regular(k):** A read that does not overlap with a write returns the result of one of the latest k completed writes. A read that overlaps with a write returns either the result of one of the latest k completed writes or the eventual result of one of the overlapping writes.

3. **k-atomic(k):** A read operation returns one of the values written by the last k preceding writes in an order consistent with real time (assuming there are k initial writes with the same initial value).

Allowing the system to return one of the last k written values allows for the system to be achieve higher availability than in implementations based on strict quorum systems, while ensuring good worst-case guarantees even when the network is asynchronous.

Discussion Traditional notions of safe, regular and atomic semantics [85] can be seen as a special case of the corresponding k semantics, where the maximum allowed staleness is 1. The properties offered by k-safe, and k-atomic registers are not comparable with those offered by a P-random register; however, the k-regular register can be seen as a special case of the P-random register that can be implemented even if the network is asynchronous.

For database applications, researchers have considered weakened consistency semantics for increased concurrency [81, 107]. *Epsilon consistency* attempts to increase availability by allowing query accesses to see some temporary inconsistencies in the data; however these inconsistencies are bounded and the system converges to a global serializability [107]. Krishnamurthy et al. [81] present a technique called *bounded ignorance* for increasing the concurrency in database applications, where the application may be unaware of at most N transactions.

File systems storing multiple data objects can try to enforce stronger consistency semantics, where operations are applied in the same order also across different objects [24, 75]. These requirements can be relaxed to allow for higher concurrency [21,67,111,105]. For example, TACT [122,123,124,125] is a toolkit that allows for the consistency level of the system to change and can be used to specify various kinds of weakened semantics. PRACTI [21] allows to implement easily different consistency semantics across different data

objects under different network topologies. In this dissertation, we focus on the consistency semantics defined with respect to a single data object.

Chapter 8

Implementing k-atomic registers

In this chapter we provide an implementation for the k-atomic register. Section 8.1 introduces the notion of a k-quorum system. In Section 8.2.1 we propose a single-writer implementation of the k-atomic register tolerating only benign failures. In Section 8.2.2 we show how to implement a single-writer k-atomic register that handles Byzantine failures. Finally, in Section 8.3 we show how single-writer k-atomic registers can be used to implement multiple-writer registers.

8.1 k-quorum constructions

A k-quorum system consists of (i) an underlying strict quorum system, $(\mathcal{R}, \mathcal{W})$, and (ii) a staleness parameter l that is the bound on the staleness allowed.

Definition. A k-quorum system over a set $\mathcal{P}$ is defined as a triple $\big(\mathcal{R}, \mathcal{W}, l\big)$, where $\mathcal{R} \subset 2^{\mathcal{P}}$ is the set of read quorums, $\mathcal{W} \subset 2^{\mathcal{P}}$ is the set of write quorums, and l is a staleness parameter.

Read and write operations Read operations in k-quorums are similar to reads in the traditional quorum systems. At a high level, the reader r contacts a quorum of servers $R \in \mathcal{R}$ and returns the latest retrieved value.

Writes are a little different. Writes in k-quorums may be written to any subset of $\mathcal{P}$, such that the servers contacted during l consecutive writes form a write-quorum $W \in \mathcal{W}$. We henceforth call the set of servers contacted during a particular write a *partial-write-quorum*.

To tolerate f Byzantine servers failures, we require that for any $R \in \mathcal{R}$, and $W \in \mathcal{W}$, $|R \cap W| \geq 3f + 1$ and $|R|, |W| \leq (n - f)$.

8.2 A single-writer register

We now present a single-writer-multiple-reader implementation of a k-atomic register using k-quorums.

Model We consider a system consisting of a set $\mathcal{P}$ of n *servers*. The servers can communicate with the clients (readers and writers) over authenticated point-to-point channels. Each server (or node) can crash and recover. We assume that servers have access to a stable storage mechanism that is persistent across crashes. We place no bound on the number of non-Byzantine faulty servers and, when considering Byzantine faults in Section 8.2.2, we assume that there are no more than f Byzantine servers in the system. The network is assumed to be asynchronous. The clients are assumed to be correct, and the writers are assumed to have at most one outstanding write.

8.2.1 Handling benign failures

We first present a protocol that tolerates only benign server failures. The single-writer-multiple-reader protocol for implementing a k-atomic register is shown in Figure 8.1.

```
// Protocol for the single-writer
static k := 0;
static ts := 0;

void Write(value)
begin
    ts := ts+1;
    k := k+1;

    Find an available partial-write-quorum, W_k, such that:
            ∃W ∈ 𝒲 : W ⊆ ⋃_{i=k-l+1}^{i=k} W_i.

    Write (value, ts, PW) to W_k, where PW = ⋃_{i=k-l+1}^{i=k-1} W_i
    Wait for acknowledgments from W_k
    return
end


// Protocol for a reader
int Read()
begin
    Find an available read quorum, R, and read from R.
    (v,ts, PW) := value with the largest time stamp.

    Write back the value, (v,ts, PW) to a partial-write-quorum, W_r, such that
        ∃W ∈ 𝒲 : W ⊆ PW ∪ W_r
    Wait for acknowledgments from the partial write quorum, W_r.

    return (v, ts, PW).
end
```

Figure 8.1: k-atomic register implementation.

Write operation The write operation in k-quorums differs from the write operation in normal quorum systems and requires the value to be written to only a subset of the write quorum known as a partial-write-quorum. To ensure a bound on staleness, we require that any l partial-write-quorums used for successive write operations must collectively contain a write quorum.

Formally, let W_i be the partial-write-quorum used in the i^{th} write. We require that

$$\forall i : \exists W \in \mathcal{W} \; such \; that \; W \subseteq \bigcup_{j=i-l+1}^{i} W_j$$

The protocol for write is shown in Figure 8.1. During the i^{th} write, the writer writes to each of the servers in W_i: (i) the value – v, (ii) the timestamp – ts, and (iii) the set PW of servers accessed in the previous $(l-1)$ writes.

Read operation To perform a read operation, the reader contacts a read quorum of servers to collects their responses and chooses the value with the latest time stamp (v, ts_{hst}, PW). The reader then writes back the tuple (v, ts_{hst}, PW) to a set of servers W' such that $\exists W \in \mathcal{W} : W \subseteq PW \cup W'$.

The protocol for a read is shown in Figure 8.1. Since a read quorum always intersects with one of the previous l partial-write-quorums, a read is guaranteed to return one of the l latest written values irrespective of the behavior of the scheduler.

Analysis To prove that the protocols achieve k-atomic(l) semantics, we show the existence of a linearized schedule for reads and writes such that each read returns the value written by one of the previous l writes.

Theorem 16. The read and write protocols shown in Figure 8.1 provide k-atomic(l) semantics.

Proof. Let *written-time* denote the global time instance when a value that is being written reaches a partial-write-quorum. We will order the reads and writes such that:

- All writes are ordered as if they instantaneously take place at their written-time.
- A read which returns a value (v, ts, PW), which was written at timestamp ts, can be scheduled at any time between

1. The written-time, τ_{ts} of the value returned, (v, ts, PW); and
2. The written-time of the next l^{th} write, $(v', ts + l, PW')$. i.e. before τ_{ts+l}.

It is easy to see that such an ordering satisfies the requirements of k-atomic(l) semantics. We need to show that the ordering can be achieved in a manner consistent with local history. The scheduling of writes is trivial, because the written-time of a write occurs between the time a write has begun and before the write ends.

We now show, by contradiction, that reads can also be scheduled. Suppose that the read interval did not overlap with the interval $[\tau_{ts}, \tau_{ts+l})$. There are two cases:

1. The read finishes before τ_{ts}: This senario is not possible, because a read completes only after performing a write-back on the value. Therefore a read can end only after the written-time of the value it returns.
2. The read begins after τ_{ts+l}. Consider the union of the partial write quorums for l previous writes – $W_{ts+1} \cup W_{ts+2} \cup \ldots \cup W_{ts+l}$. From the definition of a partial-write-quorum and the fact that any read quorum intersects with a (complete) write quorum, it follows that the reader has received a value from at least one server in $W_{ts+1} \cup W_{ts+2} \cup \ldots \cup W_{ts+l}$.

   However, since the reader chooses the highest timestamp received, a read that starts after τ_{ts+l} cannot return a value written before τ_l, which is a contradiction.

□

8.2.2 Handling Byzantine servers

We now show a protocol that implements a k-atomic(l) register in the presence of Byzantine faults. We assume that the system consists of n servers of which there are no more than f are Byzantine.

```
static Reading = ∅
static current_data[1..k];

while( true ) {
  (msg, sender) = receiveMessage();

  if( msg instanceof READ_REQUEST)
    Reading ∪= {sender};
    send current_data to sender.

  else if( msg instance of STOP_READ )
    Reading = Reading \ {sender};

  else if( msg instance of WRITE )
    // say msg is WRITE ⟨Tuple[ts_new, ..., ts_new − k + 1]⟩
    if( ts_new.ts > current_data[1].ts )
      current_data[1..k] = Tuple[ts_new, ..., ts_new − k + 1];
      send ACK(ts_new) to sender;
      forward current_data to all in Reading.
    else
      send ACK(ts_new) to sender;
}
```

Figure 8.2: k-quorum protocol for servers.

Server-side protocol Figure 8.2 shows the server-side protocol. Each server s maintains in the structure *current_data* information about the last write the server knows of, as well as the $l-1$ writes that preceded it. READ_REQUEST messages are handled using a "listeners" pattern [95]. The sender of such requests is added to s's *Reading* set, which contains the identities of the clients with active read operations at s. A read operation r is active at s from when s receives r's READ_REQUEST to when it receives the corresponding STOP_READ. On receipt of a WRITE message, s acknowledges the writer.

Then, if the received information is more recent than the one stored in *current_data*, s updates *current_data* and forwards the update to all the clients in *Reading*; otherwise, it does nothing.

```
static ts := 0;
static Tuple[];
void Write(value v)
begin
    ts := ts + 1;
    h = hash(Tuple[ts − 1, ..., ts − k + 1]);
    // E is the set of servers NOT used for the previous k − 1 writes
    E = P \ ⋃_{j=ts−k+1}^{j=ts−1} W_j
    Tuple[ts] = (v, ts, E, h);
    delete Tuple[ts − k] to save space

    Find a set PW, such that: |PW ⋃ ∪_{j=ts−k+1}^{j=ts−1} W_j| = Q_w
    send WRITE⟨Tuple[ts, ..., ts − k + 1]⟩ to all servers in PW.

    // wait for acknowledgements
    W_ts = ∅
    do
      recv ACK(ts) from serv
      W_ts = W_ts ∪ {serv}
    until ( |∪_{j=ts−k+1}^{j=ts} W_j| ≥ Q_w − f )
    return
end
```

Figure 8.3: k-quorum write protocol tolerating f Byzantine servers.

Writer's protocol Figure 8.3 shows the client-side write protocol. Each write operation affects only a small set of servers, called a *partial write quorum*, chosen by the writer so that the set of its last l partial write quorums forms a complete write quorum. The information sent to the servers contains not just a new value and timestamp, but also additional data that will help readers distinguish legitimate updates from values fabricated by Byzantine servers. Specifically, the writer sends l tuples to each server in the partial write quorum – one for each of its last l writes. The tuple for the i-th of these writes includes: i) the value v_i; ii) the corresponding timestamp ts_i; iii) the set E_i of servers

that were not written to in the last $l-1$ writes preceding i; and iv) a hash of the tuples of the $l-1$ writes preceding i. The write ends once the set of servers from which the writer has received an acknowledgment during the last l writes forms a complete write quorum[1].

Thus, the value, timestamp, E, and hash information for write i are not only written to i's partial write quorum, but also to the partial write quorums used for the next $l-1$ writes. By the end of these l writes this information will be written to a complete write quorum which is guaranteed to intersect any read quorum in at least $3f+1$ servers.

Reader's protocol Figure 8.4 shows the client-side read protocol. To perform a read operation, the reader contacts a read quorum of servers and collects from each of them the l tuples they are storing. The goal of the read operation is twofold: first, to identify a tuple t_i representing one of the last l writes, call it i, and return to the reader the corresponding value v_i; second, to write back to an appropriate partial write quorum (one comprised of servers not in E_i) both t_i and the $l-1$ tuples representing the writes that preceded i—this second step is necessary to achieve k-atomic(l) semantics.

The read protocol computes three sets based on the received tuples. The *Valid* set contains, of the most recent tuples returned by each server in the read quorum, only those that are also returned by at least f other servers. The tuples in this set are legitimate: they cannot have been fabricated by Byzantine servers.

[1]Byzantine servers may never respond. The writer can address this problem by simply contacting f additional nodes for each write while still only waiting for a partial quorum of replies. For simplicity, we abstract these details in giving the protocol's pseudocode.

```
received[] // stores the responses from servers
CandidateValues // holds the set of candidate values
Read()
begin
  choose a read quorum R;
  send READ_REQUEST to servers in R;

  received[i] = null, 1 ≤ i ≤ |R|;
  CandidateValues = ∅
  // receive values from all the servers in R
  while( |{i : received[i] ≠ null}| < |R| )
  begin
    receive Tuple[ts_s,...,ts_s − k + 1] from server s;
    received[s] = Tuple[ts_s,...,ts_s − k + 1];
    if( isValid(Tuple[ts_s,...,ts_s − k + 1]) )
      add Tuple[ts_s] to the set CandidateValues;
  end

  // try to choose a value: if unsuccessful, wait for more responses
  ts_highest = LargestTimestamp(received);
  tryChoosing( );
  while( value_chosen == null )
  begin
    receive Tuple[ts_s,...,ts_s − k + 1] from server s;
    if (ts_s ≤ ts_highest )
      received[s] = Tuple[ts_s,...,ts_s − k + 1];
      tryChoosing( );
  end

  send STOP_READ to servers in R;

  // write back the chosen value to a  partial−write−quorum
  Find a partial−write−quorum, PW, suitable for value_chosen.
  send WRITE⟨ value_chosen ⟩ to PW;
  wait for acks from PW;

  return  value_chosen;
end

void tryChoosing( )
begin
  (1) Fresh = { Tuple[ts_s,...,ts_s − k] ∈ Received | ts_s is one of the 2f+1 largest
      time−stamped entries in Received received from different servers}
  (2) Valid = { Tuple[ts_s,...,ts_s − k] ∈ Received | Tuple[ts_s] occurs in the
      responses of at least f + 1 servers}
  (3) Consistent = { Tuple[ts_s,...,ts_s − k] ∈ Received | the hash, h, in Tuple[ts_s]
      matches hash(Tuple[ts_s − 1,...,ts_s − k])}
  (4) if ( Valid ∩ Fresh ∩ Consistent ≠ ∅ )
        value_chosen = v ∈ Valid ∩ Fresh ∩ Consistent, with the largest timestamp.
end
```

Figure 8.4: k-quorum read protocol tolerating f Byzantine servers.

The *Consistent* set also contains a subset of the most recent tuples returned by each server s in the read quorum. For each tuple t_s in this set, the reader has verified that the hash of the $l-1$ preceding tuples returned by s is equal to the value of h stored in t_s.

The *Fresh* set contains the $2f+1$ most recent tuples that come from distinct servers. Since a complete write quorum intersects a read quorum in at least $2f+1$ correct servers, legitimate tuples in this set can only correspond to recent (i.e. not older than l latest) writes.

The intersection of these three sets includes only legitimate and recent tuples that can be safely written back, together with the $l-1$ tuples that precede them, to any appropriate partial write quorum. The reader can choose any of the tuples in this intersection: to minimize staleness, it is convenient to choose the one with the highest timestamp.

Protocol Correctness We first prove that the read protocol, shown in Figure 8.4, only returns values that are: (i) actually written by the writer (as opposed to an arbitrary value generated by a Byzantine server), and (ii) are not more than l writes old.

Lemma 16. If the algorithm in Figure 8.4 returns a value, Tuple$[ts, \ldots, ts - l + 1]$, then the writer must have written Tuple$[ts, \ldots, ts - l + 1]$.

Proof. The algorithm returns a value, Tuple$[ts, \ldots, ts-l+1]$, only if the value belongs to *Valid* $\cap$ *Consistent*. For Tuple$[ts, \ldots, ts - l + 1]$ to be present in *Valid*, the latest of the $l+1$ values – Tuple$[ts]$ – has to be reported by at least $f+1$ different servers. Since at least one of these servers is correct, it follows that Tuple$[ts]$ was written by the writer.

Moreover, since Tuple$[ts, \ldots, ts - l + 1]$ also belongs to *Consistent*, the the hash in Tuple$[ts]$ has to matches hash(Tuple$[ts - 1, \ldots, ts - l + 1]$). Therefore, the history of the previous $l - 1$ writes is also correct and was written by the writer. □

Lemma 17. The set *Fresh* never contains a value that is more than l writes old.

Proof. The intersection between a read and a write quorum consists of at least $3f + 1$ servers. Hence, among the servers responding there are at least $3f + 1$ servers who have "seen" one of the latest l writes. At least $2f + 1$ of these are correct and have a timestamp greater than or equal to the l-th latest write that occurred before the read has begun. Since the timestamp at a correct server monotonically increases, the correct servers in the intersection will never return a value that is more than l writes old.

Since there are at least $2f + 1$ correct servers who never report a value more than l writes old, then the $2f + 1$ latest values received from different servers, *Fresh*, will never contain a value that is more than l writes old. □

Theorem 17. The single-writer Byzantine k-quorum read protocol in Figure 8.4 never returns a value that is has not been written by the writer.

Proof. Follows from Lemma 16. □

Theorem 18. The single-writer Byzantine k-quorum read protocol in Figure 8.4 never returns a value that is more than l writes old.

Proof. The read in Figure 8.4 only returns a value that belongs to $Valid \cap Consistent \cap Fresh$. From Lemma 17, we know that the set *Fresh* can never

contain a value that is more than l writes old. Hence, a read will never return a value that is more than l writes old. □

In an asynchronous environment where there is no bound on the number of nodes failing, no protocol can provide liveness guarantees always. If the network is behaving asynchronously, or if the required number of servers is not available, then our protocols will just stall until the systems comes to a good configuration. We will now argue that if the required number of servers is accessible, and the network behaves synchronously, then our protocols will eventually terminate.

Theorem 19. If the network behaves synchronously and all non-Byzantine nodes recover and stay accessible, then the Byzantine k-quorum protocol for the writer in Figure 8.3 eventually terminates.

Proof. If network is synchronous, and the non-Byzantine nodes recover, then the writer will be able to find an accessible partial-write-quorum. On receiving the acknowledgements from all the servers in the partial-write-quorum, the writer terminates. □

Theorem 20. If the network behaves synchronously and all non-Byzantine nodes recover and stay accessible, then the Byzantine k-quorum protocol for the reader in Figure 8.4 eventually terminates.

Proof. New values from a server are allowed to overwrite old values only as long as the time stamp of the new value is at most $ts_{highest}$. Therefore values cannot be overwritten indefinitely.

Consider the situation, after the writer completes the latest write before $ts_{highest}$: say ts_{latest}[2].

When this happens all correct servers in the intersection of the read and write quorum will have a timestamp ts such that $ts_{latest} - l + 1 \leq ts \leq ts_{latest}$. The correct servers will forward the values and these values will not be overwritten by any other value[3].

Since the intersection between a read and a write quorum contains at least $2f + 1$ correct servers, eventually the reader will receive at least $2f + 1$ values that have their timestamp in the range $[ts_{latest} - l + 1, ts_{latest}]$. Consider the $(f+1)$-th largest timestamped value, v_c, received from some correct server.

Since v_c is reported by a correct server, its hashes match and therefore v_c is present in *Consistent*. Since there are no more than f faulty servers that may report higher timestamps, v_c will be present also in $Fresh$. Further, since all the $2f + 1$ highest timestamped values from correct servers lie in the range $[ts_{latest} - l + 1, ts_{latest}]$, it follows that the $(f + 1)$-th value from a correct server will be contained in the history of the first f highest-timestamped values from correct servers. Hence, v_c is also present in $Valid$. Therefore, `tryChoosing` will set *value_chosen* to a non-null value and the algorithm will terminate. □

Theorem 21. The protocols described in Figure 8.2, 8.3 and 8.4 implement k-atomic(l) semantics.

[2]If the writer has not performed a write operation with a timestamp greater than $ts_{highest}$, then ts_{latest} will be the timestamp of the last write operation. Otherwise ts_{latest} will be $ts_{highest}$.

[3] if $ts_{latest} \leq ts_{highest}$ then the writer has not written any value with a time stamp greater than ts_{latest}, so these values are never overwritten. Otherwise, if ts_{latest} is equal to $ts_{highest}$, then the values are not overwritten because we discard values with timestamps greater than $ts_{highest}$.

Proof. To prove that the protocols achieve k-atomic semantics, we show a linearized schedule of reads and writes such that every read returns one of the l previously written values. Let *written-time* denote the global time when a value that is being written reaches a partial-write-quorum. We will order the reads and writes operations as follows:

- All writes are ordered as if they instantaneously take place at their written-time.
- A read that returns a value (v, ts, E, h) can be scheduled at any time between
  1. The written-time, τ_{ts} of the value returned, (v, ts, E, h); and
  2. The written-time of the next l^{th} write, $(v', ts + l, E', h')$. i.e. before τ_{ts+l}.

It is easy once again to see that such an ordering satisfies the requirements of k-atomic semantics. We need to show that the ordering can be done in a manner consistent with the local history.

The scheduling of writes is trivial, because the written-time of a write occurs after the time the write has begun and before the write ends.

We now show, by contradiction, that reads can also be scheduled. Suppose that the read interval did not overlap with the interval $[\tau_{ts}, \tau_{ts+l})$. There are two cases:

1. The read finishes before τ_{ts}: This scenario is not possible, because a read has to write-back the value. Therefore, a read can end only after the written-time of the value it returns.

2. The read begins after τ_{ts+l}: From Lemma 17, any read that starts after τ_{ts+l} cannot return a value as old as τ_{ts}, which contradicts the assumption that the value returned has a timestamp ts.

□

8.3 Supporting multiple writers

We present a construction for a m-writer, multi-reader register with using single-writer, multi-reader registers with k-atomic semantics. Using k-atomic registers, our construction provides $\big((2m-1)(k-1)+m\big)$-atomic semantics.

The single-writer registers can be constructed using the k-quorum protocols from Section 8.2.1 if servers are subject to crash and recover failures, or using the construction from Section 8.2.2 if servers are subject to Byzantine failures.

To differentiate between the single-writer and multiple-writer read and write operations, we assume that the single-writer implementation provides the following two operations.

1. *val* sw-kread(wtr): returns one of the k latest written values, by the writer *wtr*.

2. sw-kwrite(wtr, *val*): writes the value *val* to the k-quorum system. It can only be invoked by the writer *wtr*

Further, we assume that the read and write availability of the single-writer k-quorum system is $a_{sr} = 1 - \epsilon_{sr}$ and $a_{sw} = 1 - \epsilon_{sw}$ respectively.

8.3.1 Multiple writer construction

The multi-writer construction uses m instances of the single-writer k-atomic register, one for each writer w_i. It uses approximate vector timestamps to compare writes from different writers. Each writer w_i, $1 \leq i \leq m$, maintains a local virtual clock lts_i, which is incremented by 1 for each write so that its value equals the number of writes performed by writer w_i.

At a given time, let $g\vec{t}s$ be defined by

$$\forall i : g\vec{t}s[i] = lts_i$$

where the equality holds at the time of interest. The vector $g\vec{t}s$ represents the global vector timestamp and it may not be known to any of the clients or servers in the system. The read and write protocols are shown in Figure 8.5.

Write Operation To perform a write operation, the writer first performs a read to obtain the timestamp information about all the writers (lines 5-6). Since the registers used are k-atomic, the received timestamp information is guaranteed to be no more than k writes old for any writer.

A writer wtr_i executing a write would calculate (lines 9-10) an approximate vector timestamp $a\vec{t}s$, whose i-th entry is equal to lts_i and whose remaining entries can be at most k older than the local time stamps of the entries at the time the write operation was started. Let $gts^{\vec{beg}}$ and $gts^{\vec{end}}$ denote the global timestamps at the start and end of the write. Then,

$$a\vec{t}s[i] = gts^{\vec{end}}[i]$$

$$a\vec{t}s[j] > gts^{\vec{beg}}[j] - k$$

$$gts^{\vec{end}} \geq gts^{\vec{beg}}$$

```
1  static ltsi = 0;
2  void mw-write( writeri , val )
3  begin
4    ltsi ++
5    for j = 1 to m
6      ⟨valj, ts⃗j⟩ = sw-read( writerj )
7
8    // Estimate the approx time-stamp
9    ∀j ≠ i : at⃗s[j] = max1≤p≤m { ts⃗p[j] }
10   at⃗s[i] = ltsi ;
11
12   sw-write( writeri , ⟨val, at⃗s⟩ )
13 end
14
15 ⟨val, ts⟩ mw-read( )
16 begin
17   for j = 1 to m
18     ⟨valj, ts⃗j⟩ = sw-read( writerj )
19
20   Reject = ∅
21   for i = 1 to m
22     for j = 1 to m
23     if( ts⃗j < ts⃗i || (ts⃗j[i] < ts⃗i[i] − k) )
24       Reject = Reject ∪ {⟨valj, ts⃗j⟩}
25
26   return any ⟨valj, ts⃗j⟩ ∉ Reject
27 end
```

Figure 8.5: Multi-writer k-quorum protocols.

The writer then writes the value, val, along with the timestamp $\vec{ats}$ to the single-writer k-atomic register for the writer.

Read operation To perform a multi-writer read operation, a reader reads from all the m single-writer k-atomic registers. Because of the k-atomicity of the underlying single-writer implementation, each of these m responses is guaranteed to be one of the k latest values written by each writer. However, if some writer has not written for a long time, then the value could be very old when considering *all* the writes in the system. Finding the latest value among these m values is difficult because the approximate timestamps are not totally ordered.

The reader uses elimination rules (lines 20-24) to reject values that can be inferred to be older than other values. This elimination is guaranteed to reject any value that is more than $\big((2m-1)(k-1)+m\big)$ writes old. Finally, after rejecting old values, the reader returns any value that has not been rejected.

8.3.1.1 Protocol correctness

We now analyze the protocol in Figure 8.5 to give a bound on the staleness.

Lemma 18. If a writer w_i performs a write, beginning at the (global) time $\vec{gts}^{beg}$ and ending at $\vec{gts}^{end}$, with a (approximate) timestamp $\vec{t}$, then

$$\vec{t} \leq \vec{gts}^{end}; \quad \vec{t}[i] = \vec{gts}^{end}[i]; \text{ and}$$

$$\forall j : \vec{t}[j] \geq \vec{gts}^{beg}[j] - k + 1$$

Proof. The k-quorum implementation of the single-writer system for writer w_j, guarantees that any sw-read for writer w_j will return one of the k latest values written by w_j. Thus, during the initial read phase, the writer w_i will read one of the last k timestamp values used by w_j. Hence $\vec{gts}^{end}[j] \geq \vec{t}[j] \geq \vec{gts}^{beg}[j]-k+1$ for all j.

Moreover, the writer w_i always sets the i^{th} coordinate of the computed vector timestamp to his local virtual timestamp lts_i (line 10). Therefore $\vec{t}[i] = \vec{gts}^{end}[i]$. □

Lemma 19. Let $\langle val_j, \vec{ts}_j \rangle$ be one of the m values read in lines 17-18. If a writer, say i, has performed $2k$ writes after $\langle val_j, \vec{ts}_j \rangle$ has been written (and before the read starts), then $\langle val_j, \vec{ts}_j \rangle$ will be rejected in lines 20-24.

Proof. Let $\langle val_i, \vec{ts_i} \rangle$ be the value read from writer i. Let $\vec{gts_j^{beg}}$,$\vec{gts_j^{end}}$ and $\vec{gts_i^{beg}}$,$\vec{gts_i^{end}}$ denote the global timestamp at the beginning and end of the writes for $\langle val_j, \vec{ts_j} \rangle$ and $\langle val_i, \vec{ts_i} \rangle$. Also, let $\vec{gts_{read}^{beg}}$ be the timestamp when the read is started.

Since writer i has performed at least $2k-1$ writes after writing $\langle val_j, \vec{ts_j} \rangle$ we have

$$\vec{gts_{read}^{beg}}[i] \geq \vec{gts_j^{end}}[i] + 2k$$

Also, from the k-atomic properties of the single writer system, we know that

$$\begin{aligned} \vec{ts_i}[i] = \vec{gts_i^{end}}[i] &> \vec{gts_{read}^{beg}}[i] - k \\ \Rightarrow \vec{ts_j}[i] \leq \vec{gts_j^{end}}[i] &\leq \vec{gts_{read}^{beg}}[i] - 2k \\ &< \vec{gts_i^{end}}[i] - k = \vec{ts_i}[i] - k \end{aligned}$$

Hence $\langle val_j, \vec{ts_j} \rangle$ will be added to Reject in line 24. □

Theorem 22. The multi-writer read protocol never returns a value that is more than $\big((2m-1)(k-1)+m\big)$ writes old.

Proof. Let $\langle val_j, \vec{ts_j} \rangle$ be the value returned by the read protocol.

The writer j cannot have written more than $k-1$ writes after $\langle val_j, \vec{ts_j} \rangle$ (and before the read begins). From Lemma 19 it follows that each of the remaining $(m-1)$ writers could have written no more than $2k-1$ writes after the write for $\langle val_j, \vec{ts_j} \rangle$ (and before the read begins). Hence, $\langle val_j, \vec{ts_j} \rangle$ can be at most $\big(1+(k-1)+(m-1)(2k-1)\big)$ writes old. □

Lemma 20. At least one of the m received values is not rejected.

Proof. A value $\langle val_j, \vec{ts_j} \rangle$ is rejected if either of the following two condition applies:

- Condition (i): $\exists i : \vec{ts_j} < \vec{ts_i}$
- Condition (ii): $\exists i : \big(\vec{ts_j}[i] < \vec{ts_i}[i] - k\big)$

Among all the m values received, consider the value $\langle val_l, \vec{ts_l} \rangle$ whose write started last. It cannot be rejected by Condition (i) because all other writes started before it.

Consider the write for any other value $\langle val_j, \vec{ts_j} \rangle$. Since this write has started before $\vec{gts_l^{beg}}$, it follows that

$$\vec{gts_l^{beg}}[j] \geq \vec{gts_j^{beg}}[j] = \vec{ts_j}[j] - 1$$

Since when writer l performs a read to estimate $ts_l[j]$ it is guaranteed to receive a value no older than k writes,

$$\vec{ts_l}[j] \geq \vec{gts_l^{beg}}[j] - k + 1$$

$$\Rightarrow \vec{ts_l}[j] \geq \vec{ts_j}[j] - k$$

Thus $\langle val_l, \vec{ts_l} \rangle$ will not be rejected by Condition (ii) either. □

Theorem 23. The multi-writer protocol described in Figure 8.5 provides $\big((2m-1)(k-1)+m\big)$-atomic semantics.

Proof. To prove that the multi-writer protocols achieve $\big((2m-1)(k-1)+m\big)$-atomic semantics, we show that there exists a serialized schedule of reads and writes, consistent with the local history, where each read operation returns a value written by one of the last $\big((2m-1)(k-1)+m\big)$ write operations.

Write operations are scheduled in the order of the approximate timestamp. Let W_i and W'_j be two write operations by writers i and j, and with approximate timestamps $\vec{ts_i}$ and $\vec{ts_j}$ respectively. Then, W_i is scheduled before W_j if $\vec{ts_j}[i]$ is greater than or equal to $\vec{ts_i}[i]$. W_j is scheduled before W_i if $\vec{ts_i}[j]$ is greater than or equal to $\vec{ts_j}[j]$. If neither of these two conditions hold, then the two writes are considered concurrent and can be scheduled in either order.

We schedule the reads as follows: any read that returns a value $\langle val_i, \vec{ts_i} \rangle$, written during the write W_i, is scheduled to occur at some point after write W_i but within the next $\big((2m-1)(k-1)+m\big)$ write operations.

It is easy to see that this ordering satisfies the requirements of $\big((2m-1)(k-1)+m\big)$-atomic semantics. We need to show that such an ordering can be done in a manner consistent with local history.

The ordering of the write operations is consistent with each other, because it is consistent with the local timestamps at the writers.

Suppose that a read operation that returns a value $\langle val_i, \vec{ts_i} \rangle$, which was written during the write W_i, cannot be scheduled in a manner consistent with the local history. Then, either (i) the read operation ends before W_i begins, or (ii) the read operation begins only after there are more than $\big((2m-1)(k-1)+m\big)$ write operations that succeed W_i. We show that both these cases result in a contradiction:

1. Suppose the read ends before W_i begins. Note that mw-read only returns a value that has been read from sw-read in line 6. Thus, if mw-read were to finish before write i starts, this would contradict the assumption that the underlying single-writer implementation satisfies k-atomicity.

2. Suppose the read operation begins after there are more than $\big((2m-1)(k-1)+m\big)$ write operations that succeed W_i. It follows, from Theorem 22, that the read operation could not return the value $\langle val_i, \vec{ts_i}\rangle$ written during the write W_i, which is more than $\big((2m-1)(k-1)+m\big)$ writes old.

□

Availability of a Multi-writer System We now estimate the availability of the multi-writer system, assuming that the underlying single-writer k-quorum system has read and write availability of $a_{sr} = 1-\epsilon_{sr}$ and $a_{sw} = 1-\epsilon_{sw}$ respectively.

Each multi-writer write operation involves reading from all the m single-writer k-atomic registers and writing to one single-writer register. Hence the write availability of the multi-writer register, a_{mw}, is at least $(a_{sr})^m a_{sw}$. This is a conservative estimate because we are assuming that, when the network is synchronous, we treat finding a read quorum and finding a partial-write-quorum as independent events. In practice, however, the fact that a particular number of servers (size of read quorum) are up and accessible only increases the probability of being able to find an accessible partial-write-quorum.

Moreover, if the m underlying single-writer k-quorum systems are implemented over the same strict quorum system, then the potential read quorums that can be used for all the m systems will be the same.[4] Thus, we can use the same read quorum to perform all the m read operations. In this

[4]The partial-write-quorums could still be different, if the writers have chosen different partial-write-quorums in the past.

case, either all reads are available with probability a_{sr} or all reads fail with probability ϵ_{sr}. Hence the probability of the multi-writer write succeeding is at least $a_{sr}a_{sw}$.

$$a_{mw} \geq a_{sr}a_{sw} \geq 1 - \epsilon_{sr} - \epsilon_{sw}$$

To perform a multi-writer read, our read protocol performs m reads from the m single-writer k-atomic registers. Thus, along similar lines, we can argue that the availability a_{mr} is at least ${a_{sr}}^m$. Using the same underlying strict quorum system for all the m single-writer registers, we can achieve an availability of

$$a_{mr} = a_{sr} = 1 - \epsilon_{sr}$$

8.3.2 A lower bound

We now show a lower bound on the staleness of a multi-writer register implementation built using single-writer k-atomic registers. Specifically, we show that by using k-atomic single-writer registers as primitives for a multi-writer register with m writers, one cannot provide a consistency that is better than $\big((2m-1)(k-1)+1\big)$-atomic guarantees. Thus, the implementation in Section 8.3 is less than m stale values away from the optimal.

Since we are interested in a multi-writer solution that has the same availability as the underlying single-writer register, we should rule out solutions that require a write in the multi-writer register to invoke multiple write operations of the single-writer register. In other words, a write operation in the multi-writer system should be able to terminate successfully if a read quorum and a partial write quorum of the single-writer system are available. We require that a read quorum be available because otherwise writers would be forced to write independently of each other with no possibility for one writer

to *see* other writes. We do not require that a read and a write quorum be available at the same time. So, without loss of generality, we assume that the implementation uses only m single-writer registers, one for each writer. The implementation of a write operation of a the multi-writer register can issue a write operation to the issuing writer's register but not to the other writers' registers; it can also issue read operations to any of the m registers. The read operations on the multi-writer register can only issue read operations on the single-writer registers.

In our lower bound proof, we assume that writers execute a full-information protocol in which every write includes all the history of the writer, including all the values it ever wrote and all the values it read from other writers. If the lower bound applies to a full-information protocol, then it will definitely apply to any other protocol, because a full-information protocol can simulate any other protocol by ignoring portions of the data read. Also, we assume that a reader and a writer read all single-reader registers in every operation, possibly multiple times; a protocol that does not read some registers can simply ignore the results of such read operations.

For a writer wtr, we denote with $v_{wtr,i}$ the i'th value written by wtr. If a client reads $v_{x,i}$, then it will also read $v_{x,j}$, $j \leq i$. We denote with $\vec{ts}_{wtr}$ a vector timestamp that captures the writer's knowledge of values written to the system. $\vec{ts}_{wtr}[u]$ is the largest i for which wtr has read a value $v_{u,i}$. In what follows, we will simply denote values with their indices. So, we will say that a writer writes a vector timestamp instead of writing values whose indices are less than or equal to the indices in the vector timestamp.

We now describe a scenario where a reader would return a value that happens to be $\big((2m-1)(k-1)+1\big)$ writes old.

Consider a multi-writer read operation, where the timestamps for all the m values that the reader receives are similar—specifically, the timestamps

$$Rcvd = \left\{ \begin{array}{c} \langle k-1, 0, 0, \ldots, 0\rangle, \\ \langle 0, k-1, 0, \ldots, 0\rangle, \\ \langle 0, 0, k-1, \ldots, 0\rangle, \\ \vdots \\ \langle 0, 0, 0, \ldots, k-1\rangle \end{array} \right\}$$

where the timestamp for the value received from the i-th writer contains information up to the $(k-1)$-th write by that writer, but only contains information about the 0-th write for all remaining writers.

Since all the m timestamp values are similar, the reader would have no reason to choose one value over the other. Let us assume, without loss of generality, that the reader who reads such a set of timestamp returns the value with the timestamp

$$\langle k-1, 0, 0, \ldots, 0\rangle$$

written by the first writer.

We now show a set of writes to the system wherein the value returned would be $\big((2m-1)(k-1)+1\big)$ writes old. The writes to the system occur in four phases.

In Phase 0, each of the m writers performs a write operation such that the writer's entry in the corresponding timestamp reads 0. For the sake of this discussion, the non-positive values stored in the other entries of the timestamp are irrelevant. We refer to this write as the 0-th write.

In Phase 1, writer 1 – whose value is being returned by the read – performs $(k-1)$ writes. During each of these writes, the reads of the k-atomic register of other writers returns their 0-th write. The timestamp vector associated with each of these writes is shown in Figure 8.6.

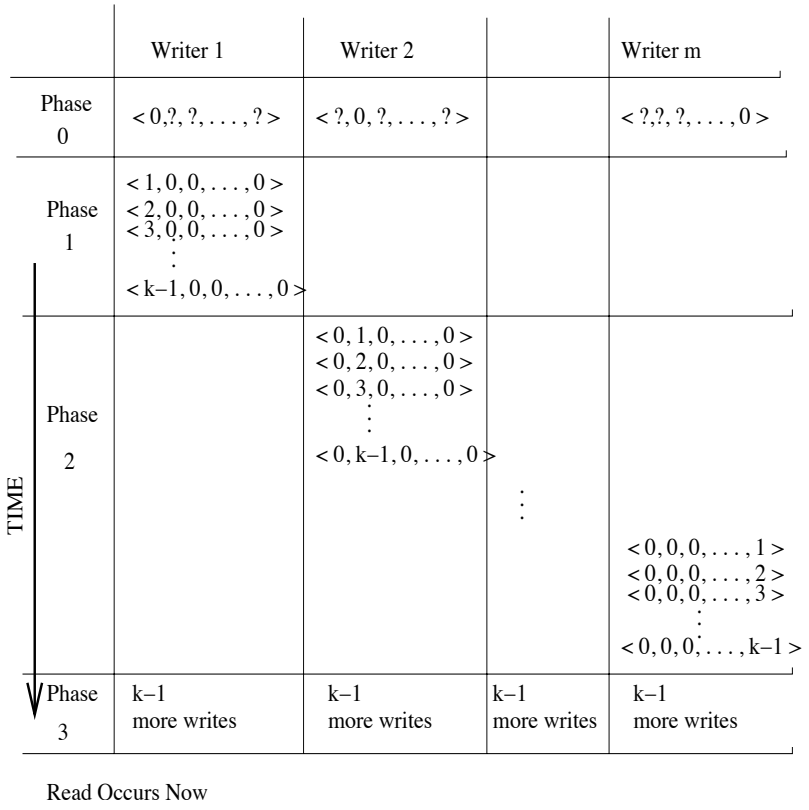


Figure 8.6: Write ordering in the multi-writer k-quorum system.

In Phase 2, each of the remaining $(m-1)$ writers perform $(k-1)$ writes. Since the underlying single-writer system only provides k-atomic semantics, also during this phase all reads to the underlying single-writer system returns the 0-th write for that writer. Hence the timestamp vector associated with these writes would be as shown in Figure 8.6.

At the end of Phase 2, each writer has performed $k-1$ writes. The total number of writes performed in this phase is $(m-1)(k-1)$.

Finally, in Phase 3, each writer performs another $k-1$ writes. There are a total of $m(k-1)$ writes in this phase. The exact timestamps associated

with these writes are not important.

At the end of Phase 3, the multi-writer read takes place. Since the underlying single-writer system only provides k-atomic semantics, all the reads to the underlying single-writer system during the read are only guaranteed to return a value which is not any older than the $(k-1)$-th write. Thus *Rcvd* could be the set of values received by the reader where the reader chooses

$$\langle k-1, 0, 0, \ldots, 0 \rangle$$

which is $\big(1 + (m-1)(k-1) + m(k-1)\big)$ writes old.

Chapter 9

Conclusion

In this dissertation, we look at distributed register abstractions and abstractions of communication channels where there is a mismatch between the strength of the assumptions used in their implementations and that of the assumptions that prevail in the environment where they are used. We develop alternative implementations for communication abstractions and implement new register abstractions to address these mismatches:

- We provide an alternative implementation of an authorized channel using secret sharing techniques instead of digital signatures.
- We provide an alternative implementation of a verifiable channel using MACs instead of digital signatures.
- We provide alternative implementations for private channels and authenticated channels among a group of n processes that use just $O(\log^2 n)$ keys at each process instead of $O(n)$ keys.
- We implement the k-atomic register which allows for a higher availability when the system is synchronous while ensuring a definite bound on staleness even asynchronous conditions.

Bibliography

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In *Proceedings of ACM SIGOPS symposium on Operating Systems Principles*, pages 59–74. ACM Press, 2005.

[2] Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Byzantine disk paxos: optimal resilience with byzantine shared memory. In *Distributed Computing*, pages 387–408. Springer-Verlag, April 2006.

[3] Ittai Abraham, Gregory V. Chockler, Idit Keidar, and Dahlia Malkhi. Wait-free regular storage from Byzantine components. *Information Processing Letters*, 101(2):60–65, 2007.

[4] Marcos Kawazoe Aguilera and Sam Toueg. Failure detection and randomization: A hybrid approach to solve consensus. *SIAM Journal of Computing*, 28(3):890–903, 1999.

[5] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. On the availability of non-strict quorum systems. In *Proceedings of the International Symposium on Distributed Computing*, pages 48–62. Springer-Verlag, 2005.

[6] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Byzantine and multi-writer k-quorums. In *Proceedings of the International Symposium on Distributed Computing*, pages 443–458. Springer-Verlag, 2006.

[7] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Bounded wait-free implementation of optimally resilient Byzantine storage without (unproven) cryptographic assumptions. In *Proceedings of the International Symposium on Distributed Computing*, pages 443–458. Springer-Verlag, 2007.

[8] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Bounded wait-free implementation of optimally resilient Byzantine storage without (unproven) cryptographic assumptions. Technical Report TR-07-32, University of Texas at Austin, Department of Computer Sciences, July 2007.

[9] Amitanand S. Aiyer, Lorenzo Alvisi, and Rida A. Bazzi. Lightweight writeback for Byzantine storage systems. Technical Report TR-07-13, University of Texas at Austin, Department of Computer Sciences, March 2007.

[10] Amitanand S. Aiyer, Lorenzo Alvisi, Rida A. Bazzi, and Allen Clement. Matrix Signatures: From MACs to Digital Signatures. In *Proceedings of the International Symposium on Distributed Computing*, pages 16–31. Springer-Verlag, 2008.

[11] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, Mike Dahlin, Jean-Philippe Martin, and Carl Porth. BAR fault tolerance for cooperative services. In *Proceedings of ACM SIGOPS symposium on Operating Systems Principles*, pages 45–58, New York, NY, USA, 2005. ACM.

[12] Amitanand S. Aiyer, Lorenzo Alvisi, Allen Clement, and Harry C. Li. Information-theoretically secure Byzantine paxos. Technical Report

TR-07-21, University of Texas at Austin, Department of Computer Sciences, May 2007.

[13] Amitanand S. Aiyer, Lorenzo Alvisi, and Mohamed G. Gouda. Key grids: A protocol family for assigning symmetric keys. In *Proceedings of the IEEE International Conference on Network Protocols*, pages 178–186, Santa Barbara, CA, Nov 2006.

[14] Noga Alon and Joel H. Spencer. *The Probabilistic Method*. John Wiley, New York, 1991.

[15] Lorenzo Alvisi, Evelyn Tumlin Pierce, Dahlia Malkhi, Michael K. Reiter, and Rebecca N. Wright. Dynamic Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 283–292, 2000.

[16] Rida Bazzi and Gil Neiger. The possibility and the complexity of achieving fault-tolerant coordination. In *Proceedings of the annual symposium on Principles of Distributed Computing*, pages 203–214, New York, NY, USA, 1992. ACM.

[17] Rida A. Bazzi and Yin Ding. Non-skipping timestamps for Byzantine data storage systems. In *Proceedings of the International Symposium on Distributed Computing*, pages 405–419. Springer-Verlag, 2004.

[18] Rida A. Bazzi and Yin Ding. Bounded wait-free f-resilient atomic Byzantine data storage systems for an unbounded number of clients. In *Proceedings of the International Symposium on Distributed Computing*, pages 299–313. Springer-Verlag, 2006.

[19] Rida A. Bazzi and Gil Neiger. Simplifying fault-tolerance: providing the abstraction of crash failures. *Journal of the ACM*, 48(3):499–554, 2001.

[20] Rida Adnan Bazzi. *Automatically increasing fault tolerance in distributed systems*. PhD thesis, Georgia Institute of Technology, Atlanta, GA, USA, 1995.

[21] Nalini Belaramani, Mike Dahlin, Lei Gao, Amol Nayate, Arun Venkataramani, Praveen Yalagandula, and Jiandan Zheng. Practi replication. In *Proceedings of the conference on Networked Systems Design & Implementation*, pages 59–72, Berkeley, CA, USA, 2006. USENIX Association.

[22] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In *CRYPTO '96: Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology*, pages 1–15, London, UK, 1996. Springer-Verlag.

[23] Josh Cohen Benaloh and Jerry Leichter. Generalized secret sharing and monotone functions. In *CRYPTO '88: Proceedings of the 8th Annual International Cryptology Conference on Advances in Cryptology*, pages 27–35, London, UK, 1990. Springer-Verlag.

[24] Philip A. Bernstein and Nathan Goodman. The failure and recovery problem for replicated databases. In *Proceedings of the annual symposium on Principles of Distributed Computing*, pages 114–122, New York, NY, USA, 1983. ACM.

[25] Kenneth P. Birman. Replication and fault-tolerance in the isis system. *SIGOPS Operating Systems Review*, 19(5):79–86, 1985.

[26] Kenneth P. Birman and Thomas A. Joseph. Reliable communication in the presence of failures. *ACM Transactions on Computer Systems*, 5(1):47–76, 1987.

[27] Matt Bishop. *Computer Security*. Addison-Wesley, 2002.

[28] Gabriel Bracha. Asynchronous byzantine agreement protocols. *Information and Computation*, 75(2):130–143, 1987.

[29] Gabriel Bracha and Sam Toueg. Asynchronous consensus and broadcast protocols. *Journal of the ACM*, 32(4):824–840, 1985.

[30] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. The primary-backup approach. In *Distributed systems (2nd Ed.)*, pages 199–216. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[31] Mike Burrows. The chubby lock service for loosely-coupled distributed systems. In *Proceedings of the Symposium on Opearting Systems Design & Implementation*, pages 335–350, Berkeley, CA, USA, 2006. USENIX Association.

[32] Christian Cachin and Stefano Tessaro. Optimal resilience for erasure-coded Byzantine distributed storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 115–124, Washington, DC, USA, 2006. IEEE Computer Society.

[33] Marco Carpentieri. A perfect threshold secret sharing scheme to identify cheaters. *Designs, Codes and Cryptography*, 5(3):183–187, 1995.

[34] Miguel Castro. *Practical Byzantine Fault Tolerance*. Ph.D., MIT, January 2001. Also as Technical Report MIT-LCS-TR-817.

[35] Miguel Castro and Barbara Liskov. Authenticated Byzantine fault tolerance without public-key cryptography. Technical Memo MIT-LCS-TM-589, MIT, June 1999.

[36] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance. In *Proceedings of the Symposium on Opearting Systems Design & Implementation*, pages 173–186. USENIX Association, 1999.

[37] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Transactions on Computer Systems*, 2002.

[38] Tushar Deepak Chandra, Vassos Hadzilacos, and Sam Toueg. The weakest failure detector for solving consensus. *J. ACM*, 43(4):685–722, July 1996.

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

[40] Peter M. Chen, Edward K. Lee, Garth A. Gibson, Randy H. Katz, and David A. Patterson. Raid: High-performance, reliable secondary storage. *ACM Computing Surveys*, 26:145–185, 1994.

[41] Wei Chen, Sam Toueg, and Marcos Kawazoe Aguilera. On the quality of service of failure detectors. *IEEE Transactions on Computers*, 51:561–580, 2000.

[42] Gregory Chockler, Rachid Guerraoui, and Idit Keidar. Amnesic Distributed Storage. In *Proceedings of the International Symposium on Distributed Computing*, pages 139–151. Springer-Verlag, 2007.

[43] Gregory Chockler, Rachid Guerraoui, Idit Keidar, and Marko Vukolic. Reliable distributed storage. *IEEE Computer*, 42(4):60–67, 2009.

[44] Allen Clement, Manos Kapritsos, Sangmin Lee, Yang Wang, Lorenzo Alvisi, Mike Dahlin, and Taylor Riche. Upright cluster services. In *Proceedings of ACM SIGOPS symposium on Operating Systems Principles*, pages 277–290, New York, NY, USA, 2009. ACM.

[45] Allen Clement, Mirco Marchetti, Edmund Wong, Lorenzo Alvisi, and Mike Dahlin. Making Byzantine fault tolerant systems tolerate Byzantine faults. In *Proceedings of the conference on Networked Systems Design & Implementation*, pages 153–168, April 2009.

[46] Brian A Coan. A communication-efficient canonical form for fault-tolerant distributed protocols. In *Proceedings of the annual symposium on Principles of Distributed Computing*, pages 63–72, New York, NY, USA, 1986. ACM.

[47] Brian A Coan. A compiler that increases the fault tolerance of asynchronous protocols. *IEEE Transactions on Computers*, 37(12):1541–1553, 1988.

[48] James Cowling, Daniel Myers, Barbara Liskov, Rodrigo Rodrigues, and Liuba Shrira. HQ replication: A hybrid quorum protocol for Byzantine fault tolerance. In *Proceedings of the Symposium on Opearting Systems Design & Implementation*, pages 177–190, Berkeley, CA, USA, November 2006. USENIX Association.

[49] Flaviu Cristian, Houtan Aghili, Ray Strong, and Danny Dolev. Atomic broadcast: From simple message diffusion to byzantine agreement. In *Information and Computation*, pages 200–206, 1985.

[50] Giuseppe DeCandia, Deniz Hastorun, Madan Jampani, Gunavardhan Kakulapati, Avinash Lakshman, Alex Pilchin, Swaminathan Sivasubramanian, Peter Vosshall, and Werner Vogels. Dynamo: Amazon's highly available key-value store. *SIGOPS Operating Systems Review*, 41(6):205–220, 2007.

[51] Whitfield Diffie and Martin E. Hellman. New directions in cryptography. *IEEE Transactions on Information Theory*, 22(6):644–654, January 2003.

[52] Shlomi Dolev, Seth Gilbert, Nancy Lynch, Alex Shvartsman, and J. Welch. GeoQuorums: Implementing atomic memory in mobile ad hoc networks. In *Proceedings of the International Symposium on Distributed Computing*, pages 306–320, October 2003.

[53] Ehab S. Elmallah, Mohamed G. Gouda, and Sandeep S. Kulkarni. Logarithmic keying. *M Transactions on Autonomous and Adaptive Systems*, 3(4):1–18, 2008.

[54] Laurent Eschenauer and Virgil D. Gligor. A key-management scheme for distributed sensor networks. In *CCS '02: Proceedings of the 9th ACM conference on Computer and communications security*, pages 41–47, New York, NY, USA, 2002. ACM.

[55] Rui Fan and Nancy Lynch. Efficient replication of large data objects. In *Proceedings of the International Symposium on Distributed Computing*, pages 75–91. Springer-Verlag, October 2003.

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

[57] Armando Fox and Eric A. Brewer. Harvest, yield, and scalable tolerant systems. In *HOTOS '99: Proceedings of the The Seventh Workshop on Hot Topics in Operating Systems*, page 174, Washington, DC, USA, 1999. IEEE Computer Society.

[58] William Gasarch. A survey on private information retrieval. *Bulletin of the EATCS*, 82:72–107, 2004.

[59] David K. Gifford. Weighted voting for replicated data. In *Proceedings of the seventh ACM symposium on Operating systems principles*, pages 150–162. ACM Press, 1979.

[60] Seth Gilbert and Nancy Lynch. Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *SIGACT News*, 33(2):51–59, 2002.

[61] Seth Gilbert, Nancy Lynch, and Alex Shvartsman. RAMBO II: Rapidly reconfigurable atomic memory for dynamic networks. In *Proceedings*

of the International Conference on Dependable Systems and Networks, pages 259–268, June 2003.

[62] Oded Goldreich. *Foundations of Cryptography: Basic Tools*. Cambridge University Press, New York, NY, USA, 2000.

[63] Shafi Goldwasser, Silvio Micali, and Ronald L. Rivest. A digital signature scheme secure against adaptive chosen-message attacks. *SIAM Journal of Computing*, 17(2):281–308, 1988.

[64] Li Gong and David J. Wheeler. A matrix key-distribution scheme. *Journal of Cryptology*, 2(1):51–59, 1990.

[65] Garth R. Goodson, Jay J. Wylie, Gregory R. Ganger, and Michael K. Reiter. Efficient byzantine-tolerant erasure-coded storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 135–144, June 2004.

[66] Rachid Guerraoui and Marko VukoliĆ. Refined quorum systems. In *Proceedings of the annual symposium on Principles of Distributed Computing*, pages 119–128, New York, NY, USA, 2007. ACM.

[67] Richard G. Guy, John S. Heidemann, Wai Mak, Thomas W. Page, Jr., Gerald J. Popek, and Dieter Rothmeir. Implementation of the Ficus Replicated File System. In *Proceedings of the Summer 1990 USENIX Conference*, pages 63–71, June 1990.

[68] Vassos Hadzilacos. Byzantine agreement under restricted types of failures (not telling the truth is different from telling lies). Technical Report 18-83, Aiken Computation Laboratory, Harvard University, 1983.

[69] Vassos Hadzilacos. *Issues of fault tolerance in concurrent computations (databases, reliability, transactions, agreement protocols, distributed computing)*. PhD thesis, Harvard University, Cambridge, MA, USA, 1985.

[70] Vassos Hadzilacos and Sam Toueg. Fault-tolerant broadcasts and related problems. In *Distributed systems (2nd Ed.)*, pages 97–145. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[71] Andreas Haeberlen, Petr Kouznetsov, and Peter Druschel. Peerreview: practical accountability for distributed systems. In *Proceedings of ACM SIGOPS symposium on Operating Systems Principles*, pages 175–188, New York, NY, USA, 2007. ACM.

[72] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Low-overhead byzantine fault-tolerant storage. In *Proceedings of ACM SIGOPS symposium on Operating Systems Principles*, pages 73–86, New York, NY, USA, 2007. ACM.

[73] James Hendricks, Gregory R. Ganger, and Michael K. Reiter. Verifying distributed erasure-coded data. In *Proceedings of the ACM Symposium on Principles of Distributed Computing*, pages 163–168. ACM Press, 2007.

[74] Maurice Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.

[75] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, 1990.

[76] Chi Ho, Danny Dolev, and Robbert Van Renesse. Making distributed applications robust. In *OPODIS'07: Proceedings of the 11th international conference on Principles of distributed systems*, pages 232–246, Berlin, Heidelberg, 2007. Springer-Verlag.

[77] Chi Ho, Robbert van Renesse, Mark Bickford, and Danny Dolev. Nysiad: practical protocol transformation to tolerate byzantine failures. In *Proceedings of the conference on Networked Systems Design & Implementation*, pages 175–188, Berkeley, CA, USA, 2008. USENIX Association.

[78] Jean-Pierre Hubaux, Levente Buttyan, and Srdan Capkun. The quest for security in mobile ad-hoc networks. In *ACM Symposium on Mobile Ad Hoc Networking and Computing*, 2001.

[79] Jiejun Kong, Petros Zefros, Haiyun Luo, and Lixia Zhang. Providing probust and ubiqutious security support for mobile ad-hoc networks. In *IEEE International Conference on Network Protocols*, 2001.

[80] Ramakrishna Kotla, Lorenzo Alvisi, Mike Dahlin, Allen Clement, and Edmund Wong. Zyzzyva: speculative byzantine fault tolerance. *SIGOPS Operating Systems Review*, 41(6):45–58, 2007.

[81] Narayanan Krishnakumar and Arthur J. Bernstein. Bounded ignorance in replicated systems. In *Symposium on Principles of Database Systems*, pages 63–74, New York, NY, USA, 1991. ACM Press.

[82] Sandeep S. Kulkarni and Bruhadeshwar Bezawada. A family of collusion resistant protocols for instantiating security. In *Proceedings of the IEEE International Conference on Network Protocols*, pages 279–288, Washington, DC, USA, 2005. IEEE Computer Society.

[83] Sandeep S. Kulkarni, Mohamed G. Gouda, and Anish Arora. Secret instantiation in ad-hoc networks. *Computer Communications*, 29(2):200 – 215, 2006. Dependable Wireless Sensor Networks.

[84] Leslie Lamport. The implementation of reliable distributed multiprocess systems. *Computer Networks*, 2:95–114, 1978.

[85] Leslie Lamport. On interprocess communication. part i: Basic formalism. *Distributed Computing*, 1(2):77–101, 1986.

[86] Susan K. Langford. Threshold dss signatures without a trusted party. In *Proceedings of International Cryptology Conference*, pages 397–409, London, UK, 1995. Springer-Verlag.

[87] Hyunyoung Lee and Jennifer L. Welch. Randomized registers and iterative algorithms. *Distributed Computing*, 17(3):209–221, 2005.

[88] Barbara Liskov and Rodrigo Rodrigues. Byzantine clients rendered harmless. In *Proceedings of the International Symposium on Distributed Computing*, pages 311–325. Springer-Verlag, 2005.

[89] Nancy Lynch and Alex Shvartsman. RAMBO: A reconfigurable atomic memory service for dynamic networks. In *Proceedings of the International Symposium on Distributed Computing*, pages 173–190, October 2002.

[90] Dahlia Malkhi and Michael K. Reiter. Byzantine quorum system. *Distributed Computing*, 11:569–578, 1998.

[91] Dahlia Malkhi and Michael K. Reiter. Secure and scalable replication in Phalanx. In *Proceedings of the IEEE Symposium on Reliable Distributed Systems*, pages 51–58, October 1998.

[92] Dahlia Malkhi, Michael K. Reiter, Avishai Wool, and Rebecca N. Wright. Probabilistic quorum systems. *Information and Computation*, 170(2):184–206, 2001.

[93] Jean-Philippe Martin. *Byzantine Fault-Tolerance and Beyond*. PhD thesis, The University of Texas at Austin, December 2006. TR-06-66.

[94] Jean-Philippe Martin and Lorenzo Alvisi. A framework for dynamic Byzantine storage. In *Proceedings of the International Conference on Dependable Systems and Networks*, Florence, Italy, June 2004.

[95] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Minimal Byzantine storage. In *Proceedings of the International Symposium on Distributed Computing*, pages 311–325, London, UK, 2002. Springer-Verlag.

[96] Jean-Philippe Martin, Lorenzo Alvisi, and Michael Dahlin. Small Byzantine quorum systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 374–383, June 2002.

[97] Alfred J. Menezes, Scott A. Vanstone, and Paul C. Van Oorschot. *Handbook of Applied Cryptography*. CRC Press, Inc., Boca Raton, FL, USA, 1996.

[98] Robert M. Metcalfe and David R. Boggs. Ethernet: distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, 1976.

[99] Dimane Mpoeleng, Paul Ezhilchelvan, and Neil Speirs. From crash tolerance to authenticated byzantine tolerance: A structured approach, the cost and benefits. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 227–236, 2003.

[100] Roger M. Needham. Cryptography and secure channels. In *Distributed systems (2nd Ed.)*, pages 531–541. ACM Press/Addison-Wesley Publishing Co., New York, NY, USA, 1993.

[101] Gil Neiger and Sam Toueg. Automatically increasing the fault-tolerance of distributed algorithms. *Journal of Algorithms*, 11(3):374–419, 1990.

[102] Gil Neiger and Mark R. Tuttle. Common knowledge and consistent simultaneous coordination. *Distributed Computing*, 6(3):181–192, 1993.

[103] Marshall Pease, Robert Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *Journal of the ACM*, 27(2):228–234, 1980.

[104] Adrian Perrig, Robert Szewczyk, Victor Wen, David E. Culler, and J. D. Tygar. SPINS: security protocols for sensor netowrks. In *Mobile Computing and Networking*, pages 189–199, 2001.

[105] Karin Petersen, Mike J. Spreitzer, Douglas B. Terry, Marvin M. Theimer, and Alan J. Demers. Flexible update propagation for weakly consistent replication. In *Proceedings of ACM SIGOPS symposium on Operating Systems Principles*, 1997.

[106] Evelyn Tumlin Pierce and Lorenzo Alvisi. A recipe for atomic semantics for Byzantine quorum systems. Technical report, University of Texas at Austin, Department of Computer Sciences, May 2000.

[107] Calton Pu and Avraham Leff. Replica control in distributed systems: An asynchronous approach. In *SIGMOD Conference*, pages 377–386, 1991.

[108] Lili Qiu, Yin Zhang, Feng Wang, Mi Kyung Han, and Ratul Mahajan. A general model of wireless interference. In *MobiCom '07: Proceedings of the 13th annual ACM international conference on Mobile computing and networking*, pages 171–182, New York, NY, USA, 2007. ACM.

[109] Pascal von Rickenbach, Stefan Schmid, Roger Wattenhofer, and Aaron Zollinger. A robust interference model for wireless ad-hoc networks. In *IPDPS '05: Proceedings of the 19th IEEE International Parallel and Distributed Processing Symposium (IPDPS'05) - Workshop 12*, page 239.1, Washington, DC, USA, 2005. IEEE Computer Society.

[110] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Communications of the ACM*, 21(2):120–126, 1978.

[111] M. Satyanarayanan, James J. Kistler, Puneet Kumar, Maria E. Okasaki, Ellen H. Siegel, and David C. Steere. Coda: A highly available file system for a distributed workstation environment. *IEEE Transactions on Computers*, 39(4):447–459, 1990.

[112] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: a tutorial. *ACM Computing Surveys*, 22(4):299–319, September 1990.

[113] Adi Shamir. How to share a secret. *Communications of the ACM*, 22(11):612–613, 1979.

[114] Cheng Shao, Evelyn Pierce, and Jennifer Welch. Multi-writer consistency conditions for shared memory objects. In Faith Ellen Fich, editor, *Distributed algorithms*, volume 2848/2003 of *Lecture Notes in Computer Science*, pages 106–120, Oct 2003.

[115] T. K. Srikanth and Sam Toueg. Simulating authenticated broadcasts to derive simple fault-tolerant algorithms. *Distributed Computing*, 2(2):80–94, 1987.

[116] Andrew S. Tanenbaum. *Computer networks: 2nd edition.* Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[117] Makoto Tatebayashi, Natsume Matsuzaki, and Jr. David B. Newman. Key distribution protocol for digital mobile communication systems. In *CRYPTO '89: Proceedings on Advances in cryptology*, pages 324–334, New York, NY, USA, 1989. Springer-Verlag New York, Inc.

[118] Robert H. Thomas. A majority consensus approach to concurrency control for multiple copy databases. *ACM Transactions on Database Systems*, 4(2):180–209, 1979.

[119] Martin Tompa and Heather Woll. How to share a secret with cheaters. *Journal of Cryptology*, 1(2):133–138, 1988.

[120] Vijay Varadharajan and Yi Mu. Design of secure end-toEnd protocols for mobile systems. In *IFIP World Conference on Mobile Communications*, pages 258–266, 1996.

[121] Haifeng Yu. Signed quorum systems. In *Proceedings of the annual symposium on Principles of Distributed Computing*, pages 246–255. ACM Press, 2004.

[122] Haifeng Yu and Amin Vahdat. Efficient numerical error bounding for replicated network services. In *VLDB '00: Proceedings of the 26th International Conference on Very Large Data Bases*, pages 123–133, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[123] Haifeng Yu and Amin Vahdat. The costs and limits of availability for replicated services. In *Proceedings of ACM SIGOPS symposium on Operating Systems Principles*, pages 29–42, New York, NY, USA, 2001. ACM.

[124] Haifeng Yu and Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Transactions on Computer Systems*, 20(3):239–282, 2002.

[125] Haifeng Yu and Amin Vahdat. Minimal replication cost for availability. In *Proceedings of the annual symposium on Principles of Distributed Computing*, pages 98–107, New York, NY, USA, 2002. ACM.

Vita

Amitanand Swaminathan Aiyer was born in Aurangabad, India. He studied in Hyderabad until Intermediate, and then received a B. Tech. in Computer Science and Engineering at the Indian Institute of Technology – Madras. He then joined the University of Texas at Austin in 2003 and received a M.S. in 2006.

Permanent address: 12-63, Veerappa Gadda,
Uppal, Hyderabad – 500039
India

This dissertation was typeset with LaTeX[†] by the author.

[†]LaTeX is a document preparation system developed by Leslie Lamport as a special version of Donald Knuth's TeX Program.