Threshold Cryptography with Tendermint Core

Powerful & secure software for the decentralized future

Master Thesis

Nathalie Froidevaux

Faculty of Science
at the University of Bern

January 2020

Prof. Dr. Christian Cachin

Cryptology and Data Security Group
Insitute of Computer Science
University of Bern, Switzerland

Abstract

Threshold cryptography provides protocols and techniques for building secure distributed systems and services that perform cryptographic operations while tolerating multiple faults and security breaches occurring at the same time. Threshold cryptosystems are operated by multiple parties which execute the cryptographic operations in collaboration and maintain privacy, availability and correctness even in the presence of some corrupted parties. The emergence and rise of blockchain technologies resulted in an increasing interest in threshold cryptography. PROTECT is an open-source platform for robust threshold cryptography and implements the most relevant protocols and techniques to maintain threshold cryptosystems. The software currently uses the BFT-SMaRt library for consensus in order to maintain a common message log. A more modern and popular implementation of the consensus protocol is provided by Tendermint, which is the leading consensus engine for building blockchains today. Tendermint's modular design allows its implementation over a generic interface without any restrictions concerning the application's programming language. This thesis is motivated by Tendermint's qualities and examines the requirements for PROTECT to implement Tendermint's consensus engine. By modeling and studying the system design and the involved communication layers of PROTECT this thesis provides a corner stone for the implementation of Tendermint's interface to upgrade PROTECT with latest blockchain-based technology.

Contents

1 Introduction

Information and communication comprise the nerve system of civilization. Consequently, the availability of reliable methods for the protection of information from intruders and adversaries is essential. *Information security* is the practice of protecting information. The primary concepts in information security are confidentiality, integrity and availability of data [11]:

Confidentiality guarantees the protection of our data from those who are not authorized to view it.

Integrity refers to the prevention of unauthorized changes or deletion of our data.

Availability is the ability to access our data when we need it.

These concepts can be compromised in various ways, e.g. by the loss of a cell phone, a wrongly addressed e-mail, a person looking over our shoulder while we type a password, power loss, network attacks, or other system problems. In past centuries, information security and in particular cryptology was important mainly for diplomatic and military communication, and mechanisms to protect confidentiality were invented in the early days of communication, such as the 'Caesar cipher' 50 B.C. The First World War encouraged greater use of code making and breaking, and machines were developed to make encryption more sophisticated. A famous example is the Enigma Machine, an encryption device engineered by the Germans at the end of the First World War and adopted by Nazi Germany in the Second World War. The successful reconstruction of the machine's mechanism implied an exemplary advantage for the Allied nations and illustrates the historical significance of information security and cryptography. Since the end of the 20th century, the rapid advancements in telecommunications, computing hardware and software and the connection of computers through the internet made information security ubiquitous.

Cryptography is the practice and study of techniques for secure communication in the presence of malicious third parties. Until 1976, the only kind of encryption publicly known was *symmetric-key cryptography*, in which both the sender and the receiver hold the same key to encrypt and decrypt a message. Ideally, a different key is used for each ciphertext exchanged and for each distinct pair of communicating parties, as well. Hence, the problem of the secure key exchange and the number of keys required are significant drawbacks in this technique [13]. For instance, the base settings of the Enigma machine's rotors and rings were changed every day. This list of settings were documented in a codebook and copies of it were distributed to each unit operating in the same network. In a paper published in 1976, Diffie and Hellman [19] proposed the concept of *public-key cryptography*, also called *asymmetric-key cryptography*. In a public-key system each network member calculates two different but mathematically related keys, a private and a public key. With the knowledge of the public key it is unfeasible to compute the corresponding private key. Hence, the public key can be freely distributed, while the private key must

be kept secret. A receiver's public key can be used by any communication party to encrypt a message which the receiver can decrypt using its corresponding private key. This technique doesn't necessitate any key exchange and the number of required keys in the communication system is reduced to one key pair per communicating party. One of the first known examples of high-quality public-key algorithms is *RSA* (a solution presented by R. Rivest, A. Shamir and L. Adleman), and has been among the most widely used [28]. Other algorithms include ElGamal encryption and various elliptic-curve techniques. Public-key cryptography is also used to implement *digital signature schemes*. By attaching a digital signature to a message, a sender provides authentication and further guarantees the integrity of the message. In this context, the sender uses its private key to construct the digital signature, and a receiver uses the corresponding public key of the sender to verify the digital signature.

However, the most common attacks on cryptographic security mechanisms are "system attacks" where the cryptographic keys are directly exposed [16] and thus, affect the reliability and availability of the entire system. A common approach to enhance the security is a *periodic refreshment of secrets* [16], e.g. changing passwords and session-keys. Consequently, an attacker is forced to be constantly active, which increases the risk of its detection and reduces the damage of one leaked secret. Another approach to enhance the security is the *distribution of cryptographic trust* [16], where the cryptographic operations, such as signing or decryption, are performed collaboratively by a group of parties. Every party holds only a share of the private key and a number of parties (*threshold*) is required to perform a cryptographic operation. Since the private key is not held entirely by one single instance, such *threshold cryptosystems* can tolerate some faulty parties and hence, have no single point of failure. The combination of a threshold cryptosystem and the periodic refresh of the secrets results in a *proactively secure cryptosystem*, where the shares of the parties are refreshed periodically [14]. Thanks to the rise of cloud services, blockchain applications and crypto-currencies, all relying on highly available and reliable distributed systems, the design of proactively secure cryptosystems has received renewed attention.

PROTECT (*P*latform for *Ro*bust *T*hr*e*shold *C*rypt*o*graphy) [27] is an open-source software that can be used to implement proactively secure cryptosystems. It implements the most important threshold-cryptographic protocols, performs threshold-cryptographic operations and can manage confidential elements durably over long periods. PROTECT facilitates the development of distributed systems and services that tolerate multiple simultaneous faults and security breaches without loss of privacy, availability, or correctness. The PROTECT project has been presented at the *NIST Threshold Cryptography Workshop 2019* [26], which was organized by the *National Institute of Standards and Technology* (NIST) to define specifications about threshold cryptography and establish consensus on applications. PROTECT contributed by presenting a platform with robust implementations for building fault-tolerant threshold cryptosystems in asynchronous networks. This requires a reliable and resilient agreement procedure among the shareholders, since they need to maintain a common message log. An implementation of such a *consensus* protocol is provided by *BFT-SMaRt* [31], an open-source Java-library that is currently used by PROTECT as its consensus layer. BFT-SMaRt is a robust and high performing implementation facilitating decentralized consensus. Compared to earlier implementations, BFT-SMaRt provides a more modular design and can be used over an interface that offers appropriate methods to perform the message exchange. In the meantime, more recent implementations of consensus protocols have been developed using the qualities of blockchain technology. The blockchain data structure, consisting of transactions that are irreversibly linked to each other using cryptography, represents a secure and efficient way to provide consensus. Tendermint [7] offers a fault-tolerant consensus engine by a modular and language agnostic interface. The Tendermint interface is used over a socket protocol and by implementing a suitable wrapper in the desired programming language, a blockchain application benefits from Tendermint's state-of-the-art technology. Furthermore, Tendermint makes use of an efficient broadcast protocol and enables scalable networks. This is an opportunity for PROTECT to profit by Tendermint's qualities as an efficient consensus layer with high scalability. This thesis examines the implementation of Tendermint into PROTECT by breaking down the relevant communication layers and software components. In order to implement Tendermint's interface, the currently used wrapper classes and the relevant code in PROTECT are identified and used for adapting the requirements according to the projected implementation.

The theoretical background of the most important threshold-cryptographic protocols is documented in Chapter 2. This chapter also provides the background of techniques and tools for the implementation of distributed systems and details about PROTECTs' functionalities. Chapter 3 offers explicit descriptions of PROTECT's current system design, along with Tendermint's system design and interface components and the projected system design of PROTECT running with Tendermint. The examination of the involved communication layers and the relevant system components for Tendermint's implementation is given in Chapter 4 and the conclusion and discussion about future work can be found in Chapter 5.

2 Background

2.1 Threshold cryptography

A threshold cryptosystem is a cryptosystem of n parties, that may correspond to processes or servers, of which up to f are faulty. Distributed cryptosystems are typically known only for public-key cryptosystems, that make use of a key pair, consisting of a public and a private key. The basic idea of a threshold cryptosystem is to split the private key into n parts and distribute them to the n parties, so that every party holds one share of the private key. In order to use the private key for cryptographic operations, e.g. signing a message or decrypting a message, $f + 1$ parties are required to perform cryptographic operations in collaboration. Therefore, a threshold cryptosystem can tolerate up to f parties to be faulty, while remaining secure and still being able to perform cryptographic operations.

The following sections concerning threshold cryptography are mainly based on the handouts of the *Security and fault-tolerance in distributed systems* lecture at ETHZ by Cachin [14].

2.1.1 Secret sharing

Secret sharing forms the basis of threshold cryptography and its basic idea is that any f+1 points can define a polynomial of degree f.

In a (f+1)-out-of-n secret sharing scheme, a secret s is shared among n parties such that the cooperation of a least f+1 parties is needed to recover s. Any group of f or fewer parties should not get any information about s. The scheme has a set of n parties, $\{P_1, ..., P_n\}$, and the secret s as an element of a finite field $\mathbb{F}_q$, where q is the size of the finite field. Additionally, the scheme requires a dealer D, where $D \notin \{P_1, ..., P_n\}$, then:

1. D chooses uniformly at random a polynomial $g(X) \in \mathbb{F}_q[X]$ of degree f subject to $g(0) = s$.

   More precisely, D samples f random numbers $\{a_1, a_2, ..., a_f\}$ from the same finite field $\mathbb{F}_q$ as before, and $q > s, n$. Then, D generates the polynomial $g(x) = a_0 + a_1x + a_2x^2 + ... + a_fx^f$, where $a_0 = s$.

2. D generates shares s_i, where each share is simply one point on the polynomial and can be calculated as: $s_i = (x_i, y_i) = (i,\ g(i) \mod q)$, where $i = [1, 2, ..., n]$

3. D sends share s_i to party P_i.

To recover s, any f+1 shares can be used to reconstruct the polynomial $g(x)$, achieved through the Lagrange polynomial interpolation, a formula for interpolating a polynomial of a degree $\leq f$ that passes through f+1

points. The interpolation polynomial $g(x)$ of f+1 points is the linear combination of the basis polynomials:

$$g(x) = \sum_{j=1}^{f+1} l_j(x) y_j, \tag{2.1}$$

where the Lagrange coefficient $l_j(x)$ is defined as:

$$l_j(x) := \prod_{\substack{k=1 \\ k \neq j}}^{f+1} \frac{x - x_{\mathrm{k}}}{x_{\mathrm{j}} - x_{\mathrm{k}}} = \frac{(x - x_0)}{(x_{\mathrm{j}} - x_0)} \cdot \ldots \cdot \frac{(x - x_{\mathrm{j\text{-}1}})}{(x_{\mathrm{j}} - x_{\mathrm{j\text{-}1}})} \cdot \frac{(x - x_{\mathrm{j+1}})}{(x_{\mathrm{j}} - x_{\mathrm{j+1}})} \cdot \ldots \cdot \frac{(x - x_{\mathrm{f+1}})}{(x_{\mathrm{j}} - x_{\mathrm{f+1}})} \tag{2.2}$$

Once we have this polynomial, it's easy to retrieve the secret, since

$$g(0) = a_0 = s. \tag{2.3}$$

The (f+1)-out-of-n secret sharing scheme has perfect security, i.e, the shares held by every group of f or fewer parties are statistically independent of s (as in a one-time pad) [30].

2.1.2 Verifiable secret sharing (VSS)

For more complex distributed cryptographic protocols we need Verifiable secret sharing (VSS). This fault-tolerant protocol is used, if the dealer *D* also may be faulty. The VSS protocol ensures two goals:

- *D* should distribute consistent shares such that every group of parties qualified to recover the secret will recover the same value.
- There should be agreement in the sense that if some party terminates the sharing successfully, then every other correct party eventually also terminates successfully.

VSS is an important building block, for instance, in secure multi-party computation [20].

2.1.3 Distributed key generation (DKG)

In distributed key generation (DKG) protocols we don't need a trusted third party as a dealer. A threshold of honest parties contribute to the generation of the public key and the corresponding private key. These protocols guarantee secrecy in the presence of malicious contribution to the key calculation and ensure that:

- Corrupted parties cannot bias the selection of the key.
- Corrupted parties cannot learn information about the secret key.

For the common public-key cryptosystems, those based on RSA and discrete logarithms, DKG protocols have been designed and implemented.

2.1.4 Proactive security

There is a risk, that an attack spreads through an (f + 1)-out-of-n-threshold cryptosystem and over time affects more than f parties. Consequently, if an adversary combines the knowledge of more than f infected parties, he can reconstruct the secret and the cryptosystem is not secure anymore. To gain more resilience and robustness in a threshold cryptosystem, there is a protocol performing a refresh of the key shares periodically, such that if a share is leaked in one time period, it will be useless in subsequent periods. Therefore, *proactive cryptosystems* tolerate up to f corrupted parties during every period [14]. A period consists of a short refresh phase, during which the *refresh protocol* is executed by all parties, and a computation phase, where the operations of the cryptosystem are performed. Note that after a corruption

has been discovered, an infected party should be rebooted and re-initialized to prevent the corrupted party of participating in the refresh protocol.

For simplicity, we assume, parties can send private and authenticated point-to-point messages over secure communication channels. Further, we assume that the parties are *synchronized* and have access to a common clock and to a *synchronous broadcast channel* [14].

Proactive refresh tolerating passive attacks

The following proactive refresh protocol [22] achieves privacy against a passive adversary. A passive adversary follows the protocol but tries to obtain more information than he is entitled, by leaking secret information and by combining its knowledge [14].

Consider $n = \{P_1, ..., P_n\}$ parties in a discrete logarithm-based cryptosystem with private key x and the corresponding public key $y = g^x$. At the begin of the refresh phase, every party P_i holds a share of the private key $x_i = a(i) = \sum_{k=0}^{t} a_k i^k$ from the previous period. Compared to the $(f + 1)$-out-of-n secret sharing scheme described in 2.1.1, in the proactive refresh protocol, no dealer is required to distribute values. But similarly, every party P_i chooses uniformly at random a polynomial $b^{(\mathrm{i})}(X)$ of degree f from the same finite field $\mathbb{F}_q$. But in contrast to the $(f + 1)$-out-of-n secret sharing scheme, the sampled polynomials are all subject to $b^{(i)}(0) = 0$. Based on these polynomials, every party generates n shares, by simply evaluating the polynomials with the n indices. Then, the designated shares are sent to each other as private point-to-point messages, such that every party eventually holds n newly generated shares (one share was generated by the party itself, the other $n - 1$ shares were received from the other parties). Finally, by adding these n share-values up, the fresh shares of the private key x'_i for the subsequent phase are calculated. To prevent the leakage of used share-values, all variables, except x'_i, are erased before a subsequent phase starts. This loop is the major component of the refresh protocol and can be structured formally as follows:

1. New shares are generated and distributed:
   - Every party P_i chooses uniformly at random a polynomial $b^{(\mathrm{i})}(X) \in \mathbb{F}_q[X]$ of degree f subject to $b^{(i)}(0) = 0$.
   - Then, it generates shares $r_{ij} = b^{(i)}(j)$ for $j = 1, ..., n$.
   - Finally, it sends r_{ij} to P_j as a private point-to-point message.
2. The shares for the next period are computed. After each party P_i received n shares r_{ij}, for $j = 1, ..., n$:
   - P_i computes its new share in $\mathbb{Z}_q$ as $x'_i = x_i + \sum_{j=1}^{n} r_{ji}$.
   - Then it erases all variables except x'_i.

Note that the private key is not altered during the refresh protocol, since $b^{(i)}(0) = 0$. Hence, the shares in the next period refer to the same private key and the corresponding public key remains valid, too.

More robust proactive refresh

In step 1 of the refresh protocol a corrupted party may send inconsistent share values r_{ij} or it may send a value $\neq 0$. The extensions described by Herzberg et al. [22] and Gennaro et al. [21] use the mechanism of VSS to prevent this attack [14].

2.2 Byzantine fault-tolerant state machine replication

Combining the properties of *Byzantine fault-tolerance (BFT)* and the techniques of *state machine replication (SMR)* adds up to the implementation of distributed software that needs to reach a *consensus* and

guarantees correct operation in presence of less than $1/3$ faulty parties, behaving in an arbitrary way. "Reaching consensus" corresponds to reaching an agreement about a common value among distributed parties and is one of the most fundamental problems in distributed computing [15]. The ability to reach an agreement is an essential property of a distributed system to achieve overall system reliability, particularly in an *asynchronous* setting, where the communicating parties are not synchronized all the time.

2.2.1 Byzantine fault-tolerance

A reliable computer system must be able to cope with the failure of one or more of its parties. A party is considered faulty once its behaviour is no longer consistent with its specification. There are various possibilities of failures, such as a crash, that stops the execution of a process, or an omission of executing certain steps, or even arbitrary and adversarial behaviour. If a party is behaving maliciously and arbitrarily, such as sending conflicting information to different parts of a system, we have a *Byzantine failure* in our system. The problem of dealing with this type of failure is expressed abstractly as the *Byzantine generals problem* and has been discussed in the context of the implementation of reliable computer systems by Lamport, Shostak and Pease in 1982 [23]. Disseminating inconsistent information to other parties prevent a computer system from reaching a consensus. Hence, the condition of a computer system to be Byzantine fault-tolerant, is the ability to operate correctly in the presence of faulty parties, that prevent other parties from reaching a consensus. In order to guarantee Byzantine fault-tolerance of a computer system, there is a minimum number of correctly operating parties required. Considering a system with a realistic asynchronous setting, the bound of correct parties required to reach agreement is $n > 3f$, where n is the total number of parties and f the number of Byzantine faulty parties [25].

2.2.2 State machine replication

State machine replication [29] is a general technique for implementing fault-tolerant services in distributed systems. Distributed software is often structured in terms of *services* and *clients*:

Service. A service exports *operations* and is executed by one or more *servers*.

Client. A client makes a request in order to invoke the provided operations by the service.

In the context of state machine replication, a service (or a server) is defined as a *state machine*, consisting of a *state variable* and *commands*. The state variable *encode* the state of the state machine and commands will *transform* the state variable. Each command is implemented by a *deterministic program*, such that the same input result in the exact same transformation and/or output. Commands are executed when a client sends a corresponding request.

Using multiple servers avoids the problem of a single point of failure and improves the fault-tolerance. To implement a fault-tolerant service in a distributed system, a state machine, that represents the desired service, is *replicated and executed on distinct processors*. The set of these replicas form a *f fault-tolerant state machine* and satisfies its specifications as long as no more than f replicas become faulty. If each replica runs on a non-faulty processor, each starts in the same initial state and each executes the same requests in the same order, consequently, each replica will do the same actions and produce the same output. The essence for implementing a f fault-tolerant state machine is *replica coordination*. It must be guaranteed, that all replicas receive and process the same sequence of requests. This can be split into two requirements concerning relaying requests [29]:

Agreement. Every non-faulty party receives every request.

Order. Every non-faulty party processes the requests it receives in the same relative order.

The agreement requirement can be satisfied by using any protocol that allows a designated party, called the *transmitter*, to disseminate a value to some other party in such a way that:

1. All non-faulty parties agree on the same value.

2. If the transmitter is non-faulty, then all non-faulty parties use its value as the one on which they agree.

The order requirement can be satisfied by assigning unique identifiers to requests and then let the parties process requests according to a total ordering relation on these unique identifiers. Protocols for reaching consensus satisfies these requirements.

Tolerating faulty output devices

To obtain the output of a f fault-tolerant state machine, the outputs of all parties are combined. To implement a f fault-tolerant system, there cannot be one single party combining these outputs, due to the risk, that it might be faulty and prevent the system to produce the correct output. Solutions to this problem depend on whether the output of the state machine is to be used within the system or outside the system and are described by Schneider [29].

Tolerating faulty clients

Faults might result in clients making requests that cause the state machine to produce erroneous output or that corrupt the state machine so that subsequent requests from non-faulty clients are incorrectly processed. Methods to isolate the state machine from faulty clients, such as replicating the client, are provided by Schneider [29].

Reconfiguration

To implement a f fault tolerant state machine, it must be possible to remove faulty parties and to add repaired parties. Similarly, this condition must also be satisfied for copies of clients and output devices.

2.2.3 BFT-SMaRt

BFT-SMaRt [12] is an open-source Java-based library [31] implementing robust Byzantine fault-tolerant state machine replication. In contrast to earlier implementations of BFT SMR, such as PBFT [17] and UpRight [18], BFT-SMaRt was the first library that supported reconfiguration of the parties. Furthermore, BFT-SMaRt is able to tolerate a number of faults that previous implementations cannot. More major properties of BFT-SMaRt are:

Byzantine fault-tolerance. BFT-SMaRt tolerates up to $f < n/3$ Byzantine faults, in a system consisting of n parties.

Performance. By taking advantage of ubiquitous multicore architecture of servers, BFT-SMaRt achieves improved performance.

Modularity. Compared to earlier implementations, BFT-SMaRt implements a modular SMR protocol, as well as a reconfiguration and a state transfer module.

Reliability. BFT-SMaRt uses reliable and authenticated channels, using point-to-point communication implemented over TCP/IP.

BFT-SMaRt provided a safe and efficient implementation of BFT SMR, ready to use for applications using decentralized consensus. For example, it was in use to implement prototypes of coordination systems, key-value stores, a transaction processing engine for replicated databases, and more. The library is still maintained and improved according to new use case scenarios.

2.2.4 Tendermint

Tendermint [9] features a modern blockchain consensus engine called *Tendermint Core*, which provides many valuable properties:

Byzantine fault-tolerance. Tendermint guarantees the secure and consistent replication of an application, in presence of up to 1/3 arbitrarily failing components.

Blockchain technology. The blockchain technology facilitates a more modern setting of Byzantine fault-tolerance (BFT) by taking advantage of cryptographic authentication. Transactions are batched in blocks and each block contains a cryptographic hash of the previous block, making older blocks immutable.

Speed. Once a transaction is included in a block, it is immediately finalized. There is no need to wait for confirmations. Tendermint Core can have a block time on the order of 1 second and can handle thousands of transactions per second.

Modularity. While Tendermint is written in *Go*, a provided interface called *Application Blockchain Interface (ABCI)* enables the usage for applications written in any programming language.

Compared to BFT-SMaRt, which communicates over point-to-point channels implemented over TCP/IP, Tendermint Core uses a peer-to-peer *gossip protocol*. This procedure doesn't require every party to communicate with each other party and contributes to improved scalability. This property is crucial for today's need of agile and high performing distributed software. All this makes Tendermint Core an attractive and state-of-the-art consensus engine. Furthermore, Tendermint provides a blockchain framework called *Cosmos SDK* [2] that uses Tendermint Core for the decentralized consensus. Cosmos SDK is a popular tool for building blockchain applications and is already in use for several blockchain applications concerning cryptocurrencies and smart contracts.

2.3 PROTECT

PROTECT [27] is an open-source software written in Java and can be used to build secure cryptographic services in realistic asynchronous networks. The discussed threshold protocols based on secret sharing, such as DKG and Proactive refresh are implemented and make PROTECT a software for robust cryptographic operations in distributed networks.

2.3.1 Functionalities

Besides the DKG and the Proactive refresh, PROTECT provides many *share management* functionalities, such as storing, reading, deleting, restoring, disabling and enabling key shares of specific shareholders. PROTECT supports various *cryptographic functions*, based on elliptic-curve and RSA techniques, for instance Pseudorandom Functions, Oblivious Pseudorandom Functions, Elliptic-curve Diffie-Hellman Key Agreement, Signature Generation, Blinded Signature Generation and Decryption. Furthermore, utilities to encrypt and decrypt text files, to store and retrieve a secret, and to generate a CA certificate, are provided. Every shareholder is accessible to authenticated users over a web-interface or over a terminal, to perform the desired functions.

2.3.2 Configuration

In the provided *server configuration file*, the number of servers to be launched, along with their network addresses, can be defined. PROTECT comes with a default configuration for a network consisting of five servers. Furthermore, several thresholds can be configured, such as the BFT threshold.

The *client configuration file* is a list of all authenticated users and their access permissions. This allows a fine-grained permission control, such as the ability to initiate a DGK, to sign a message or to

perform varying share management functionalities. For the correct authentication of these users, their public keys and the CA certificates must be stored in the designated directories. To demonstrate the user management, PROTECT is initially equipped with three example clients, having varying permissions to three different secrets.

PROTECT comes with a set of ready-made certificates and encryption keys for each server and each example client. They are loaded when a server is launched and facilitate secure communication among the servers and authenticated clients.

2.3.3 Tunable fault-tolerance

As discussed before, the upper bound of faulty parties in a asynchronous network to hold BFT, is given by $f < n/3$, where n is the total number of parties in the network. PROTECT offers a *tunable* asynchronous model by separating between a *liveness* and a *safety* threshold.

Liveness. The ability of a system to make progress.

Safety. The guaranty for a correctly operating system.

Both properties have to be provided in order to maintain a useful distributed system. Note that it is not dramatic for a system to lose liveness temporarily, since the security is not necessarily affected. This separation of a liveness threshold f_L and a safety threshold f_S enables the following trade-off [10]:

$$Max\, f_S = n - 2 \cdot Max\, f_L - 1 \tag{2.4}$$

The impact of this trade-off is illustrated in Fig. 2.1, where the relation between the number of servers in a network (n) and the corresponding fault-tolerance is shown. $Max\, f$ refers to the "classical" threshold, i.e. $Max\, f < n/3$. For instance, a network with $n = 7$ parties can tolerate $Max\, f = 2$ Byzantine faults. On the other hand, using the trade-off in eq. 2.4 allows a higher fault-tolerance concerning security ($Max\, f_S$), compared to the "classical" threshold. Although the fault-tolerance concerning liveness ($Max\, f_L$) is sacrificed at the same time, the progression doesn't fall back substantially. Considering again a network with $n = 7$ parties, the fault-tolerance for security can be improved to $Max\, f_S = 3$ by sacrificing liveness and setting $Max\, f_L = 1$.

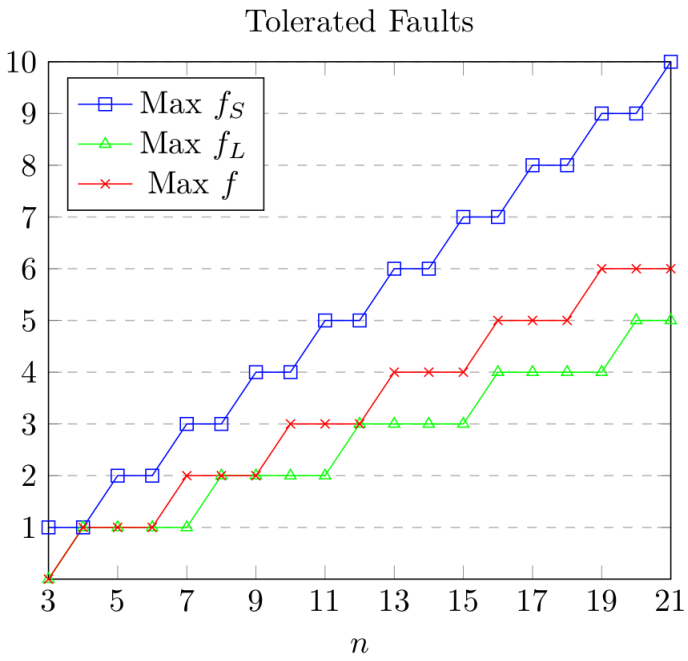

Figure 2.1: Tunable fault-tolerance [10]

2.3.4 Further development

In March 2019, the National Institute of Standards and Technology (NIST) organized a workshop [24] to define specifications about threshold cryptography and establish consensus on applications. At the *NIST Threshold Cryptography Workshop 2019*, the PROTECT project has been presented [26] as an eligible contribution, in terms of software, that provides secure cryptographic operations in asynchronous networks. PROTECT represents a valuable software that provides important functionalities for secure communication and data storage in a distributed world.

The implementation of more cryptographic operations is intended [27], such as cryptographic functions based on *Diffie-Hellman over Prime Groups* and *Bilinear Pairing of Elliptic-Curves*. But not only the development of further functionalities is of eligible interest, also the maintenance of the software itself is essential to ensure up-to-date and secure operation. On this account, this thesis tackles the update of PROTECT's *consensus layer*. Currently, PROTECT implements the earlier described BFT-SMaRt library, which performs the consensus among the parties. Due to the rise of modern and modular alternatives, such as Tendermint, it is appropriate to examine the replacement of BFT-SMaRt by Tendermint. This will contribute to the maintenance of PROTECT with the purpose to supply PROTECT with latest technology and preserve PROTECT's robustness and value.

3 Design

3.1 PROTECT with BFT-SMaRt

The major components of PROTECT are illustrated in the network abstraction in Fig. 3.1 [10]. Note that only one party (shareholder) is highlighted with the mentioned major components.

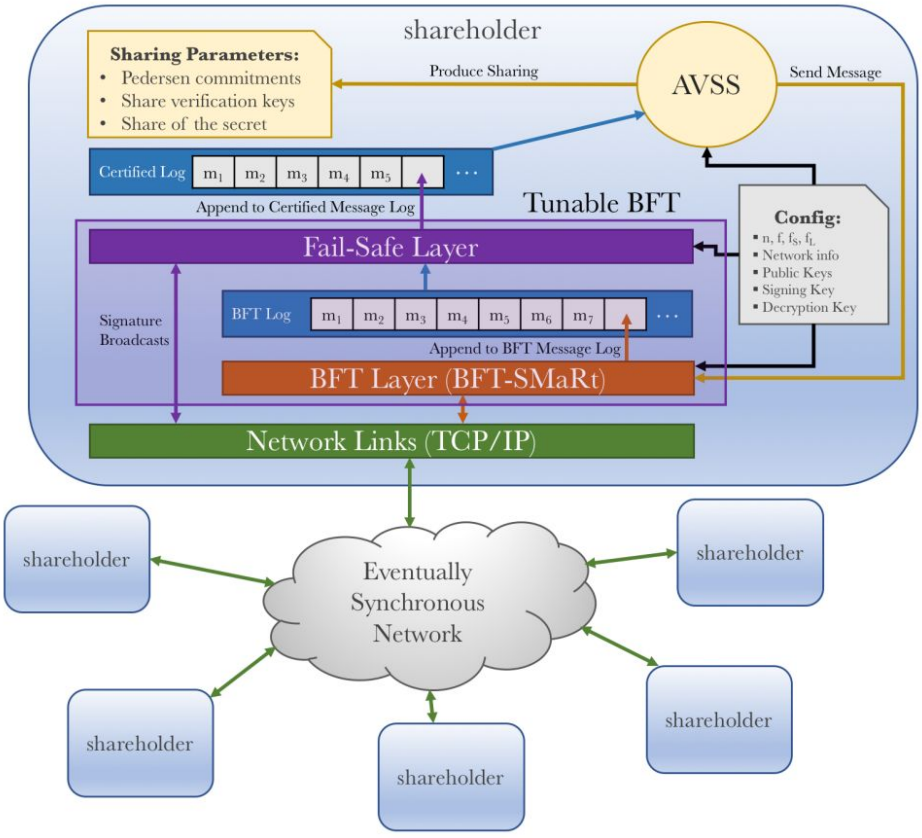

Figure 3.1: Major components of PROTECT [10]

3.1.1 Major components

The following sections provide more details about the major components and their functionalities.

AVSS application

The *AVSS application* (asynchronous verifiable secret sharing) implements the threshold protocols based on secret sharing and represents the core component of PROTECT. This component can be found in Fig. 3.1 on the top right corner. When the AVSS application is triggered to produce a message (e.g. through the initialization of the DGK protocol by a user), it broadcasts the message to the other parties by transferring it to the *BFT layer*. The state of the application is updated only by notifications of the *certified log*, which contains the commonly ordered list of certified messages.

BFT layer and BFT log

The *BFT layer* is the interface for sending and receiving atomic broadcasts of totally ordered messages. This layer receives messages from the AVSS application and communicates with the other parties over the nework layer, called *Network Links*. As long as $f < n/3$ is satisfied, the BFT-layer ensures, that all parties agree on the same ordered list of messages. This message log is represented on Fig. 3.1 as *BFT log*. This layer is currently implemented with the BFT-SMaRt library.

Fail-safe layer and certified log

The *fail-safe layer* performs extra validation of the BFT log and maintains the certified log. This log contains the same messages like the BFT log, but here the messages are certified through the extra validation step. This extra validation facilitates the tunable fault-tolerance discussed in Sec. 2.3.3 and is achieved through an additional communication step among the parties. Basically, every party checks the consistency of the BFT log, compared to the certified log and sends validations to the other parties. More precisely, whenever a new message m_k is appended to location k in the BFT log, the following steps are performed by the fail-safe layer of every party:

1. Check the consistency of all messages on positions 1 to $k-1$ in the BFT log *BL* and the certified log *CL*. Also ensure that the length of *CL* is $k-1$, as illustrated in Fig. 3.2.
2. If these conditions are met, produce a signature for the message m_k and its position k.
3. Broadcast the produced signature and the pair (m_k, k) over point-to-point links to every other party.
4. Re-broadcast every received signature to all other parties (ensuring eventual delivery of the signed pair (m_k, k)).
5. Collect $(n\text{-}f_L)$ signatures that are in mutual agreement. This collection of signatures form a *Certificate* for (m_k, k).
6. Repeat step 1.
7. If the conditions are still satisfied, append m_k to the certified message log *CL* in position k.
8. Notify the AVSS application about the availability of a new message to process.

Note that the point-to-point communication between the parties also goes through the network layer, using TCP/IP protocols, like shown in Fig. 3.1.

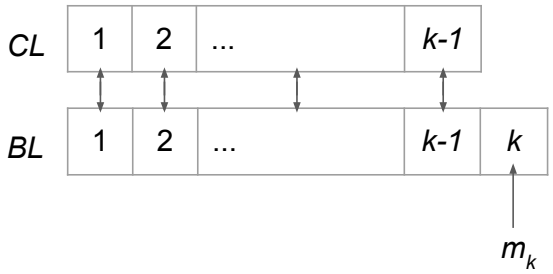

Figure 3.2: Extra validation step in the fail-safe layer

The fail-safe layer and the BFT layer (with the BFT log) together form the *tunable BFT* unit and implement the tunable fault-tolerance, as discussed in Sec. 2.3.3. This procedure also illustrates the complex and intensive communication flow in PROTECT.

3.1.2 Network configuration

The network abstraction in Fig. 3.3 depicts the default configuration of PROTECT. The provided server configuration file comes with the default setting for a local network consisting of five parties.

Considering a client-server model, PROTECT embodies the *server* for a client that wants to use PROTECT's functionalities, which are provided by the AVSS application. Each *PROTECT server* is accessible for client requests over the port 8080+*server-index*. Additionally, the PROTECT servers need a channel for point-to-point communication among each other. This is performed over the server addresses, which are also provided in the server configuration file. In Fig. 3.3 these server addresses (*protect-server-ports* 65010, 65020, 65030, 65040 and 65050) are placed on the fail-safe layer and are used for the extra validation step described in Sec. 3.1.1.

At the same time, PROTECT acts as *client* of BFT-SMaRt. Accordingly, when PROTECT is launched, it binds to port address *protect-server-port*+200, provided by BFT-SMaRt. This channel will be used to relay messages to the BFT layer. The ports *protect-server-port*+201 are used by BFT-SMaRt to execute the consensus protocol with PROTECT's requested message. The resulting agreement is replied via PROTECT's fail-safe layer.

For instance, when PROTECT *server 1* is launched, it first opens its port *65010* for server-to-server communication, secondly, it binds to port *65210*, provided by BFT-SMaRt, and finally, it opens port *8081* for client requests. More precisely, the port for client requests is not opened before the network is *BFT ready*, e.g. after three PROTECT servers are running.

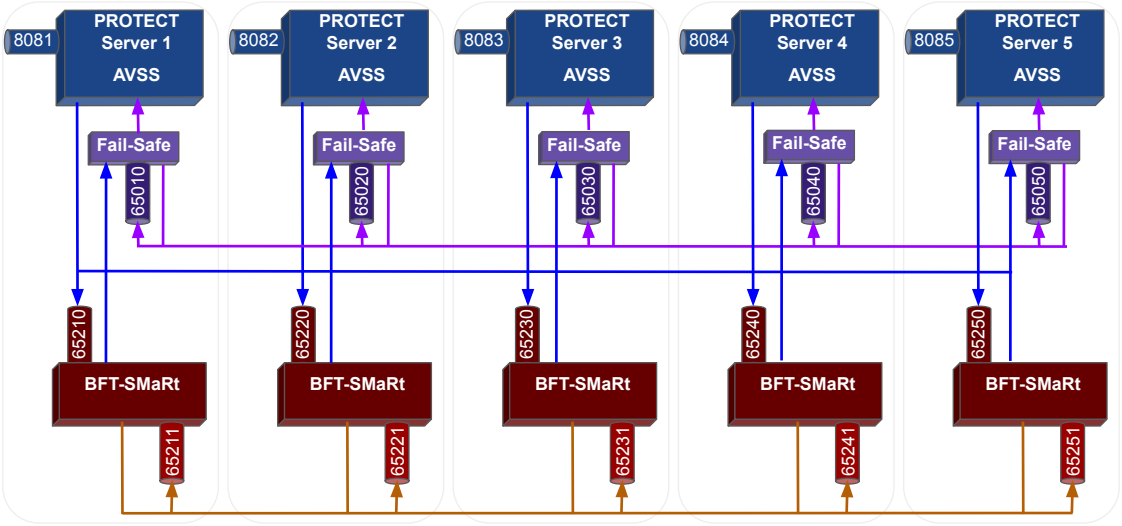

Figure 3.3: Default PROTECT network configuration

Note, that every PROTECT client will bind to every launched BFT-SMaRt server. This is due to the BFT-SMaRt design, where every client request is sent to every BFT-SMaRt party and the client will also receive a response from every party [12]. After a PROTECT server receives a client request to be executed by the AVSS application, the appropriate message is sent to the BFT-SMaRT servers. They perform the consensus protocol and get back to PROTECT via the fail-safe layer. Over point-to-point links, PROTECT processes the extra validation and certification of the message in the updated BFT log and adapts the certified log accordingly. Finally, a notification about the availability of the next message is sent to the AVSS application and updates the state of the application.

The example given in Fig. 3.4 and Fig. 3.5 demonstrates the involved classes and the relevant methods that are used when a client initializes a DKG. The sequence diagram in Fig. 3.4 shows the method calls between the initialization of the DKG and the transfer of the appropriate message to the consensus layer. The transfer is performed by the `BftChannelSender` by invoking BFT-SMaRt's `ServiceProxy`.

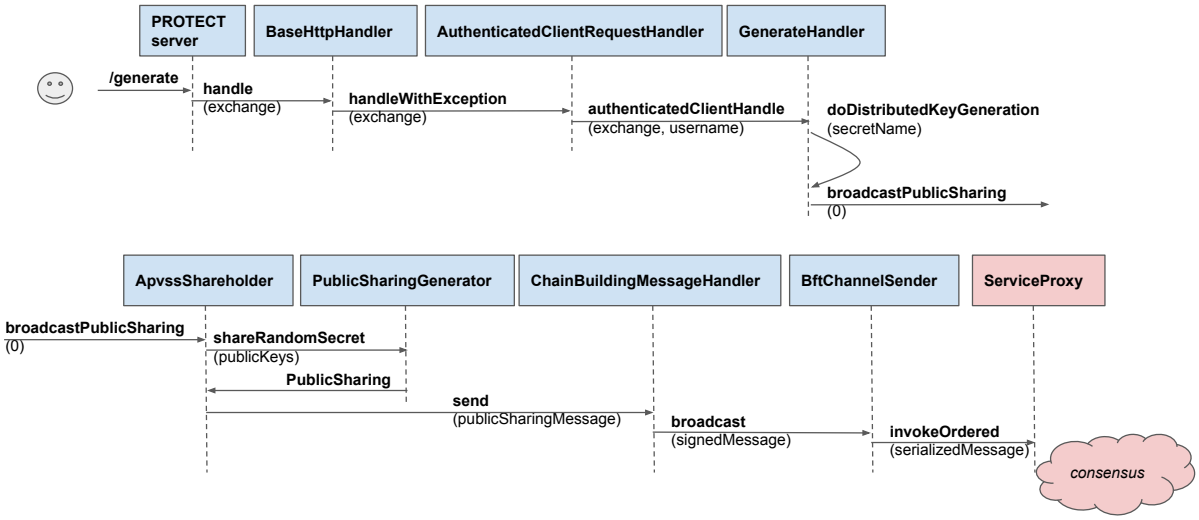

Figure 3.4: PROTECT's method calls after the initialization of a DKG

A message that is coming from the consensus layer is delivered by BFT-SMaRt's `ServiceReplica` via PROTECT's `BftListenerWrapper`. The classes shown in Fig. 3.5 handle the received message and start to perform the extra validation of the message, executed in the fail-safe layer, as described in Sec. 3.1.1.

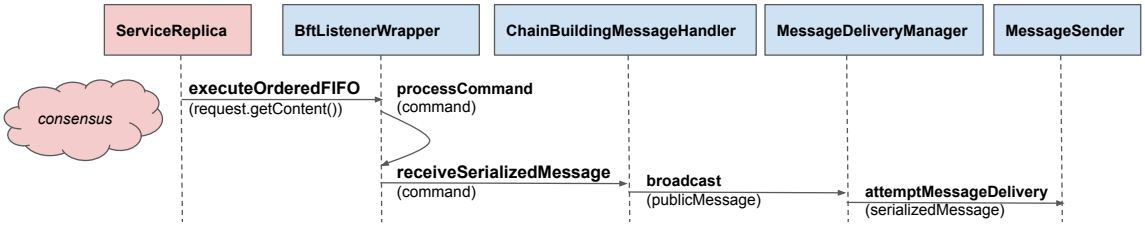

Figure 3.5: PROTECT's method calls to handle a message delivered by BFT-SMaRt

The code details of the classes in Fig. 3.4 and Fig. 3.5 are available in App. A. Besides, the classes `BftChannelSender` and `BftListenerWrapper` are discussed further in Sec. 4.1, since these two classes represent the wrapper for BFT-SMaRts interface.

3.2 Tendermint

Tendermint provides a consensus and a networking layer as a generic engine and decouples it from the details of the application. This design enables the usage of BFT operations in a modular way, without

worrying about the consensus process. Furthermore, client applications can be written in any programming language, since Tendermint's interface *ABCI* (Application BlockChain Interface) is implemented as a socket protocol.

Tendermint implements BFT using *blockchain* technology. The data structure in a blockchain offers resistance against modifications, since new *transactions* (blocks) are stored with a cryptographic hash of the previous transaction. Consequently, the transactions are linked with each other and cannot be modified, once they are stored. This property offers the maintenance of a common message log in a consensus protocol. This modern and modular implementation of a consensus engine offers a technological advantage for any sort of state machine maintaining a blockchain and is the main motivation of this thesis.

3.2.1 Tendermint Core and ABCI

Tendermint consists of two major technical components:

Tendermint Core. This is the blockchain consensus engine which ensures that the same transactions are recorded in the same order for every party.

ABCI. This generic Application BlockChain Interface enables the usage of Tendermint Core for client applications in any programming language. It is implemented as a socket protocol and can be used with a language-specific wrapper.

In order to *receive transactions* from Tendermint Core via the ABCI, a client application has to implement a wrapper, called *ABCI application*. Except for Tendermint Core, nothing else should communicate with the ABCI application, to guarantee deterministic results. Tendermint Core and the ABCI application together form a *node*, and multiple nodes form a Tendermint peer-to-peer-network, as shown in Fig. 3.6. A client can *send transactions* to be processed by Tendermint Core to any node in the network over a *Remote Procedure Call* (RPC) [6] protocol, as illustrated in Fig. 3.6. This *REST interface* can also be used for stating queries.

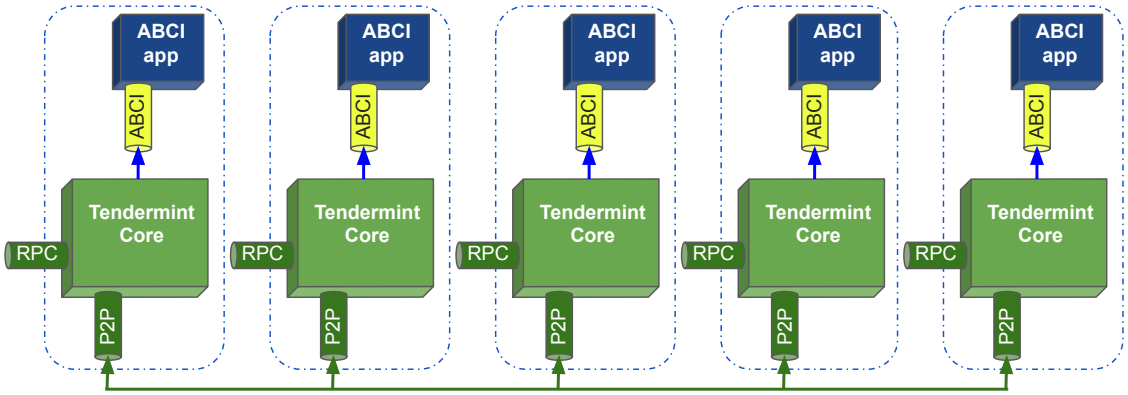

Figure 3.6: Tendermint network

The nodes communicate over a peer-to-peer network among each other, and with their ABCI application via a socket protocol, that satisfies the ABCI.

The ABCI consists of three primary *messages types*, that get delivered from Tendermint Core to the ABCI application, which replies with a corresponding response message.

CheckTx. Before a transaction is relayed to the other peers, Tendermint Core checks the validity of a transaction. More precisely, Tendermint Core checks the validity by calling the ABCI application, which replies with an approval or an error.

DeliverTx. Each transaction in the blockchain is delivered to the ABCI application with a DeliverTx message. When a transaction with this message is received, the ABCI application has to validate

it against the current state of the application, the application protocol, and the cryptographic credentials of the transaction. A validated transaction will update the application state.

Commit. This message is used to compute a cryptographic commitment to the current application state, to be placed in the next block.

Tendermint Core creates three *ABCI connections* to the ABCI application: one for the validation of transactions, before relaying it to the other peers, one for proposing blocks for the consensus procedure, and one for querying the application state. After installing Tendermint [8], several tools are provided to configure a testnet.

3.2.2 Network configuration

To run a Tendermint network, each node participating in the consensus, requires a public and a private key. Each node maintains a *config* folder, containing a file *priv_validator_key.json*, where the information about the keys are stored. Further, the public keys have to be listed in a common *genesis.json* file, to facilitate the identification among the nodes. The *config.toml* file contains the address for peer-to-peer-communication among the nodes and the address for the RPC listener.

Local testnet

A local Tendermint testnet can be initialized with a single command. Per default, a network with *four* nodes is created. The following command demonstrates the initialization for a testnet consisting of *five* nodes, where the parameter v stands for the number of *validators*:

```
$ tendermint testnet --v 5
```

This command triggers the generation of a directory called *mytestnet*, containing one folder for each node (node0, node1, node2, node3, node4). Each node maintains a config-folder, where the information about the keys, the common genesis.json file and the config.toml file can be found. Per default, all nodes have the same address configuration specified in the config.toml file:

P2P. Address to listen on for incoming peer-to-peer connections: “tcp://0.0.0.0:26656“

RPC. Address to listen on for remote procedure calls: “tcp://127.0.0.1:26657”

To avoid conflicts, it is required to change these settings by *distinctive port numbers*, for instance:

node	P2P address	RPC address
0	26656	26657
1	26659	26660
2	26661	26662
3	26663	26664
4	26665	26666

Then, every node is launched by providing a list of all peers in the network. This list contains the IDs and addresses of all nodes and the following command demonstrates how the ID of node0 is obtained:

```
$ tendermint show_node_id --home ./mytestnet/node0
```

After executing this command for every node, the launch command can be constructed. For example, to launch node0, the appropriate command is:

```
$ tendermint node --home ./mytestnet/node0 --proxy_app=kvstore
  --p2p.persistent_peers="7793b14e436a37e0d18bb3820546a3aca98e1694@localhost
  :26656,36796da0e43680b711c580c032eb10199ad58a4a@localhost:26659,
  aa5cc9c62324744f55e80eb2257690cbdd5b5544@localhost:26661,
  e957be554edbe0acb36f3888a2f8255ace7614d0@localhost:26663,
  f3888a2f8255ace7614d09e57be554edbe0acb36@localhost:26665,"
```

In order to run all other nodes, the same command has to be executed with the corresponding parameters. After starting up, the nodes will establish connections among each other and to the given example application *kvstore*, which is a sample application provided by Tendermint to demonstrate the functionality. Finally, the Tendermint nodes will be ready to receive transactions to be processed.

Distributed testnet

Tendermint provides the infrastructure to run a network in a *virtualized distributed environment*. This is realized with the help of *Docker* [3], a platform that facilitates a virtualization on the level of the operating system. This virtualization enables the simulation of a distributed system, more precisely, the usage of a virtual address space for the Tendermint nodes and the ABCI application. In order to virtualize an application with Docker, first, an application-specific *Dockerfile* is required. A Dockerfile is a script file, containing the instructions how to set up the environment for the application. Secondly, a *Docker image* is built, that consists of *read-only layers*, one per instruction specified in the Dockerfile. Finally, a *Docker container* is created by adding a *writable layer* on top of the image, where all writes to the container, such as adding, modifying or deleting data, are stored. The Docker image can be used to create multiple containers, all running with the same application, but all having their own writable layer. Docker containers are isolated from one other and communicate through well-defined channels. The process of building a Docker image out of a Dockerfile and running containers, based on the Docker image, is illustrated in Fig. 3.7. This design makes the shipping and running of applications lightweight and modular.

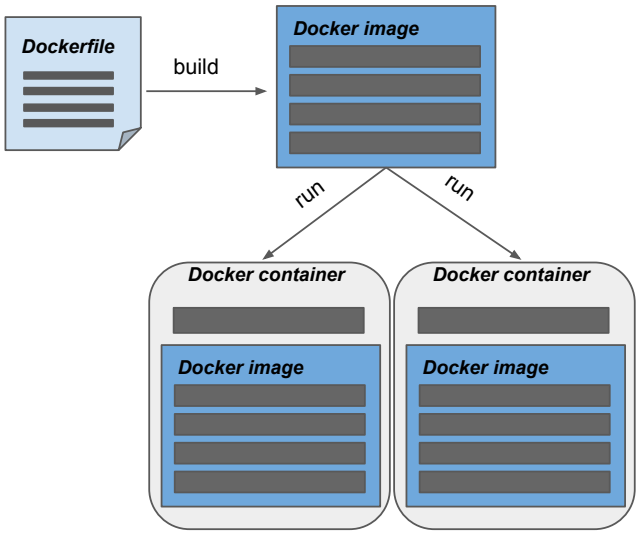

Figure 3.7: Docker container generation

To facilitate the management of Docker containers, *Docker Compose* provides a human-readable configuration file, the *docker-compose.yml*. In this file, all services of a multi-container application can be defined and enable the creation and launching of all services with the single command:

```
$ docker-compose up
```

This command will create a Docker container for every listed service.

Tendermint provides a docker-compose.yml file with configurations for a *four-node-network*, more precisely, with four listed services. This docker-compose.yml is available in App. B, as well as the relevant part of Tendermint's *Makefile* that is used to launch the network by the following command:

```
$ make localnet-start
```

Basically, this triggers the execution of the following steps:

1. Build the Docker image based on the provided Dockerfile for a Tendermint node.
2. Create folders for each node, containing the information about the keys, the genesis.json and the config.toml.
3. Execute `docker-compose up`.

The Dockerfile used in step 1 is also provided in App. B. After a container is running for every node, they start to establish the communication among each other and to the *kvstore* example application. This is specified in the Dockerfile in App. B as default application, to demonstrate the usage of Tendermint Core. The configurations in the docker-compose.yml file specify an IP-address for every node and the respective ports, port *26656* for the peer-to-peer-communication, as well as port *26657* for incoming RPC, as illustrated in Fig. 3.8. Every node establishes a connection to the ABCI application on port *26658*.

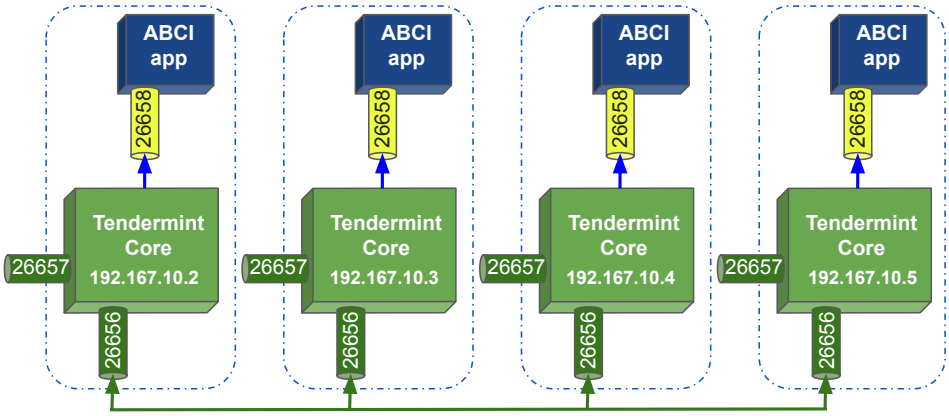

Figure 3.8: Tendermint testnet with Docker Compose

On the host machine, the nodes bind their RPC servers to ports *26657, 26660, 26662* and *26664*, hence, transactions can be sent to any of these addresses. More precisely, node0 is accessible over port 26657, node1 over port 26660, and so on. Considering the kvstore example application, that stores a key and a value, a valid transaction, addressed to node2, would be:

```
$ curl -s 'localhost:26662/broadcast_tx_commit?tx="name=satoshi"'
```

Inside the Docker environment, the transaction will be relayed to node2's address (192.167.10.4:26657) and node2 will initiate the consensus and eventually will reply to the application.

Configuration of a distributed testnet

To configure a Tendermint network with your own client application, you need to:

1. Define your application's environment with a Dockerfile. Note, that one port (e.g. *26658*) has to be exposed to make it possible for Tendermint Core to establish the communication with your ABCI application.
2. Add your application in docker-compose.yml, such that there is one application listed for every node. For instance, if you want to launch a *five-node-network*, in total *ten services* have to be listed, five for the nodes and five for the application. Docker Compose will then create one container for each service.
3. Finally, to make node-containers establishing a connection with the designated application-container when starting up, a suitable command in the service specification is required. For example, we want container *node0* to connect with container *abci0* when it's starting up, hence, we add the following line in node0's service specifications in the docker-compose.yml file:
   `command: node --proxy_app=tcp://abci0:26658`

Running a Tendermint testnet with Docker is a state-of-the-art approach to pack and run software, and the combination with Docker Compose allows efficient management of multi-container Docker applications. This motivates the usage of Docker and Docker Compose for PROTECT with the desired ability to launch multi-container networks running PROTECT and Tendermint Core.

3.2.3 Tendermint's network architecture

Tendermint provides an *appliation architecture guide* [9] containing recommendations for the architecture of a Tendermint blockchain application. The given example in Fig. 3.9 illustrates the Tendermint network running with the blockchain application *Cosmos Voyager*. The Tendermint nodes communicate over the peer-to-peer network and can receive transactions over RPC. Transactions are sent to Tendermint Core by Cosmos Voyager over a REST interface, which is provided by the *Cosmos SDK* [2]. The transactions are then committed by the Tendermint consensus and ultimately processed by the ABCI application, which is the logic that runs on the blockchain. The ABCI application must be a deterministic result of the Tendermint consensus and is also provided by the Cosmos SDK.

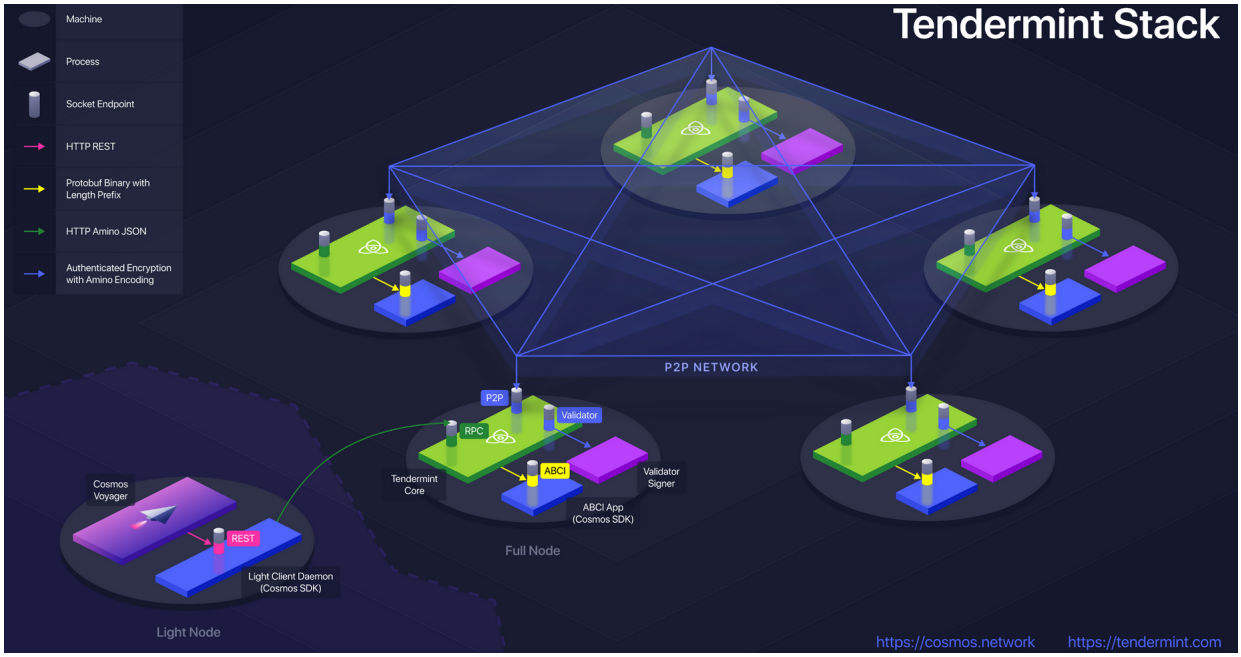

Figure 3.9: Tendermint's recommended network architecture [9]

The two components of the Cosmos SDK highlight the usage of Tendermint's interface. In order to use Tendermint Core, that offers the consensus and networking protocols, it is required for a blockchain application to supply the suitable interface wrappers. In particular, they must be able to send transactions to Tendermint Core's RPC interface and to process transactions coming from Tendermint Core's consensus.

3.3 System design

The adapted system design for PROTECT running with Tendermint is shown in Fig. 3.10 and demonstrates the network model in a Docker environment. The dashed boxes highlight two Docker containers, one running with PROTECT and one with Tendermint Core. Hence, the model illustrates a Docker environment with ten containers in total. Tendermint Core receives transactions from PROTECT and is executing the consensus over the peer-to-peer network, as shown in Fig. 3.9. The ABCI application, which is part of PROTECT, receives the committed transactions from Tendermint Core and relays them to the fail-safe layer for the extra validation procedure, as described in Sec. 3.1.1. In contrast to the network model of PROTECT running with BFT-SMaRt, illustrated in Fig. 3.3, the Docker environment enables a

virtualized address space, so that each Docker container has its own IP-address. This is demonstrated in Fig. 3.10 by the accordingly labeled PROTECT and Tendermint Core boxes. All containers are listed as services with their designated IP-address in the docker-compose.yml file, as described in Sec. 3.2.2 under *Configuration of a distributed testnet*. In order to launch the network with the containers establishing the desired connections, the different ports have to be configured as follows:

8080. PROTECT's port for client requests have to be accessible from outside of the Docker environment. The corresponding mapping of the ports is specified in the docker-compose.yml file.

26658. The port where Tendermint Core will bind on has to be exposed in PROTECT's Dockerfile.

65000. To enable server-to-server communication, all server addresses (IP-address:65000) have to be provided in PROTECT's server configuration file, as described in Sec. 2.3.2.

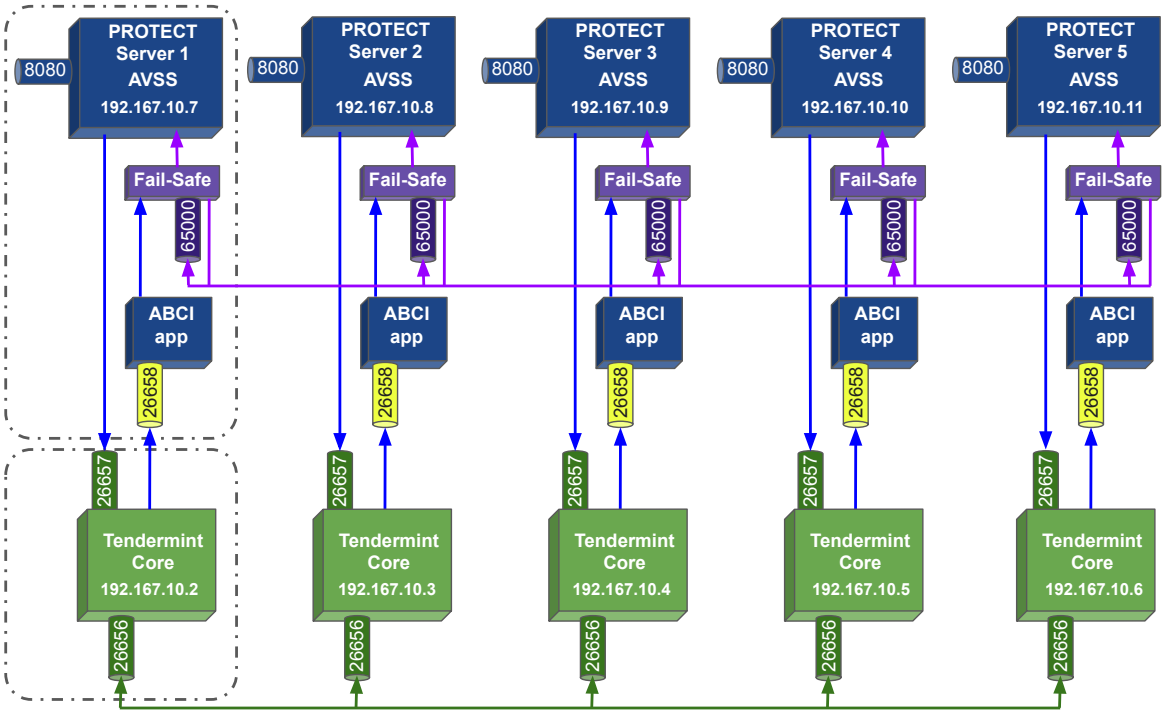

Figure 3.10: PROTECT with Tendermint in Docker environment

Corresponding to Fig. 3.9, the two arrows, one going from PROTECT to Tendermint Core and the other from Tendermint Core to the ABCI app, represent the interface between PROTECT and Tendermint Core.

Running PROTECT with Tendermint and utilising Docker and Docker Compose to configure and launch PROTECT networks, will equip PROTECT with the latest technology. Tendermint provides a modular and efficient consensus engine and Docker enables a virtualized environment for multi-container applications, that can efficiently and conveniently managed with Docker Compose.

4 Implementation

To enable the usage of Tendermint's consensus engine in PROTECT, it is necessary to carefully examine the currently used communication layers and system design in PROTECT. The Sec. 4.1 provides a detailed examination of the current implementation, more precisely, of BFT-SMaRt's interface. The implemented wrapper classes to send and receive messages are stripped down and are used to specify PROTECT's requirements to implement suitable wrappers for the use of Tendermint's interface, which is documented in Sec. 4.2. To realize the implementation, Sec. 4.3 provides the core class in PROTECT that represents the link to the wrapper classes. All this composed knowledge facilitate the actual implementation which is in progress at that time.

4.1 Examination of BFT-SMaRt's interface

The BFT-SMaRt interface consists of the following two Java classes:

`ServiceProxy`. This class represents a proxy for the client side to send messages to the parties which will perform the consensus.

`ServiceReplica`. This class replies to the client by delivering messages, after a consensus has been reached.

Since BFT-SMaRt is a Java-library, PROTECT implements this interface with method calls, executed by two wrapper classes, one instantiating a `ServiceProxy` and one a `ServiceReplica`. Consider the launch process of a PROTECT server, performed by PROTECTs' main class `ServerApplication`:

1. Load configuration
2. Load server keys
3. Load client access control
4. Set up persistent state for message broadcast and processing
5. Wait for messages and begin processing them as they arrive
6. Perform basic benchmark before starting up
7. Create message handler for the certified chain
8. Create message manager to manage messages received over point to point links

9. Create shareholder for each secret to be maintained
10. Load certificates to support TLS
11. Load client authentication keys
12. Start server to process client requests

The BFT-SMaRt interface is created during *step 7* and is illustrated in Fig. 4.1. In this step, a `ChainBuildingMessageHandler` object is generated and inside its constructor, an instance of a `BftAtomicBroadcastChannel` is created. Then, first, the wrapper class `BftChannelSender` is created, which instantiates BFT's `ServiceProxy`. Secondly, the wrapper class `BftListenerWrapper`, also instantiated by the `BftAtomicBroadcastChannel`, triggers the `ServiceReplica` instance.

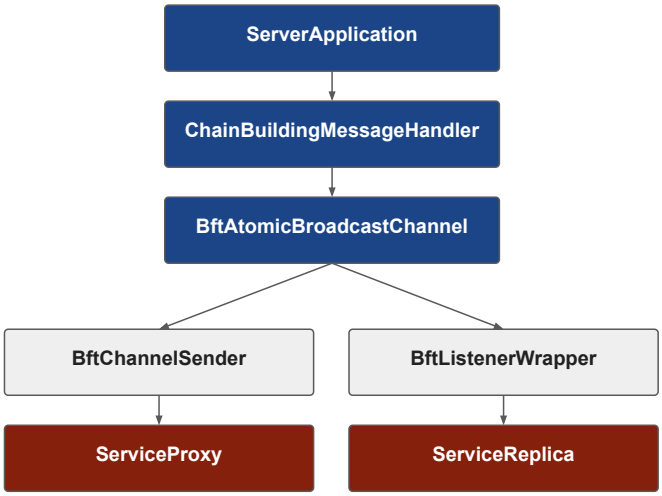

Figure 4.1: Launching BFT-SMaRt's interface

When a `ServiceProxy` is instantiated, PROTECT starts trying to establish the connection to the BFT-layer. The designated ports are created after a `ServiceReplica` is instantiated, more precisely, one port for PROTECT to bind on (*protect-server-port+200*), and one port for the consensus process (*protect-server-port+201*), as described in Sec. 3.1.2 concerning PROTECT's network configuration, and illustrated in Fig. 3.3. After the connection between PROTECT and BFT-SMaRt is established, PROTECT will handle messages to be sent to BFT-SMaRt and the messages it receives from BFT-SMaRt in the `ChainBuildingMessageHandler`. The wrapper class `BftChannelSender` is used to send a message to BFT-SMaRt over BFT-SMaRt's interface `ServiceProxy`. Messages from BFT-SMaRt are delivered by BFT-SMaRt's interface `ServiceReplica` and are processed by PROTECT's wrapper class `BftListenerWrapper`, before they are relayed to the `ChainBuildingMessageHandler`. Fig. 4.2 shows the described communication layers.

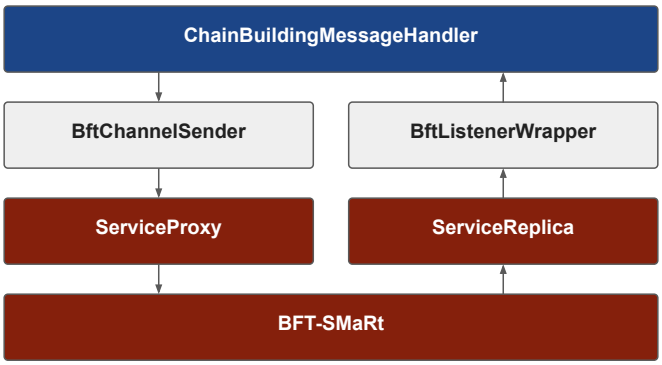

Figure 4.2: Communication layers between PROTECT and BFT-SMaRt

The wrapper classes `BftChannelSender` and `BftListenerWrapper` are described in more detail in Sec. 4.1.1 and 4.1.2.

4.1.1 Broadcasting a message to BFT-SMaRt

Messages to be sent to the BFT-layer are first *signed* by the `ChainBuildingMessageHandler` and then handed over to the wrapper `BftChannelSender` using it's `broadcast` method. The following code snippet of `BftChannelSender`'s `broadcast` method shows, that first the signed message is serialized to a byte array and then transmitted to the BFT-layer by calling `ServiceProxy`'s method `invokeOrdered(serializedMessage)`.

```
// BftChannelSender.java

   public void broadcast(SignedMessage message) {

       // Serialize message to bytes
       byte[] serializedMessage = MessageSerializer.serializeSignedMessage(message);

       // Send total ordered message
       this.serviceProxy.invokeOrdered(serializedMessage);

       // Give some time for everyone to process the message
       try {
           Thread.sleep(1000);
       } catch (InterruptedException e) {
           throw new RuntimeException("interrupted", e);
       }
   }
```

4.1.2 Receiving a message from BFT-SMaRt

To process messages received from BFT-SMaRt, the wrapper class `BftListenerWrapper` implements the method `executeOrderedFIFO(byte[] command, MessageContext msgCtx, int clientId, int operationId)`. This method is provided by the BFT-SMaRt interface `FIFOExecutable` and facilitates the delivery of requests in FIFO order to the executable instance. The method is called by the `ServiceReplica` that delivers a `command` as a byte array. The following code snippet shows that the `BftListenerWrapper` saves the received `command` and relays it to the `ChainBuildingMessageHandler` using it's method `receiveSerializedMessage(command)`.

```
// BftListenerWrapper.java

@Override
   public byte[] executeOrderedFIFO(byte[] command, MessageContext msgCtx, int
    clientId, int operationId) {
```

```
        return processCommand(command);
    }

    private synchronized byte[] processCommand(byte[] command) {
        try {
            synchronized (this.state) {
                // Save state to support recovery
                this.state.addMessage(command);

                // Process message
                this.listener.receiveSerializedMessage(command);
            }
        } catch (ClassNotFoundException | BadPaddingException |
     IllegalBlockSizeException | IOException e) {
            e.printStackTrace();
            return null;
        }
        return command;
    }
```

In order to replace BFT-SMaRt by Tendermint, suitable wrapper classes for broadcasting and receiving messages have to be implemented. The equivalent communication layers between PROTECT and BFT-SMaRt and between PROTECT and Tendermint are discussed and compared below in Sec. 4.2 and illustrated in Fig. 4.3.

4.2 Adaption of Tendermint's interface

As discussed in Sec. 3.2.1, Tendermint provides the ABCI as a generic interface. The ABCI is intended to be used for the communication between Tendermint Core and the ABCI application. Additionally, Tendermint Core can receive transactions over an RPC protocol. These components form Tendermint's interface, equivalently to BFT-SMaRt:

RPC. Transactions and queries can be sent to Tendermint Core over a REST interface.

ABCI. This is the interface between Tendermint Core and the ABCI application.

Consider Fig. 4.3, which depicts the communication layers and the message flow between PROTECT and the consensus layer. On the left part of Fig. 4.3, the communication layers between PROTECT and BFT-SMaRt are modeled, as described in Sec. 4.1. Equivalently on the right part of Fig. 4.3, the model of the communication layers between PROTECT and Tendermint are shown. PROTECT sends messages via Tendermint's interface `RPC`, using a suitable wrapper class `RPCClient`. Tendermint Core replies using the `ABCI` to an appropriate interface `ABCIListener`, provided by PROTECT. The functionalities of the `ChainBuildingMessageHandler`, such as the triggering of the message broadcast and the handling of received messages, remain unchanged.

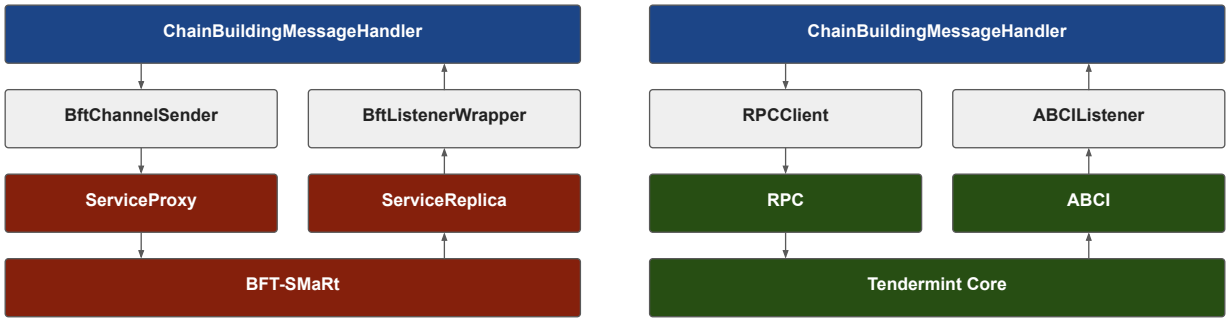

Figure 4.3: Corresponding communication layers between PROTECT and consensus layer

Consequently, the wrapper classes `RPCClient` and `ABCIListener` must provide the same functionalities to the `ChainBuildingMessageHandler` while adapting to Tendermint's interface, more precisely:

`RPCClient`. This wrapper requires to broadcast a signed message, received from PROTECT's `ChainBuildingMessageHandler`, to Tendermint Core via `RPC`. This can be implemented by using Tendermint's provided REST interface.

`ABCIListener`. This wrapper receives messages from Tendermint Core over the `ABCI`. The received messages have to be processed and forwarded to PROTECT's `ChainBuildingMessageHandler` as byte arrays. Besides, the message has to be stored to support recovery, similar to the `BftListenerWrapper` in Sec. 4.1.2.

The implementation of these wrapper classes, fulfilling the mentioned requirements, enables PROTECT the use of Tendermint's interface as consensus layer.

4.3 Realization of the interface

The core class to realize the implementation of the interface wrapper classes in PROTECT is the `BftAtomicBroadcastChannel`. As described in Sec. 4.1, the `BftAtomicBroadcastChannel` triggers the wrapper classes during the launch of a PROTECT server. Fig. 4.4 shows the corresponding relevant part of the launch process, where the wrapper classes for Tendermint's interface are created. More precisely, the `BftAtomicBroadcastChannel` instantiates the `RPCClient` and the `ABCIListener`. In contrast to the `BftListenerWrapper` which triggers BFT-SMaRt's `ServiceReplica`, illustrated in Fig. 4.1, the `ABCIListener` opens a port and waits for Tendermint Core's requests over the `ABCI`.

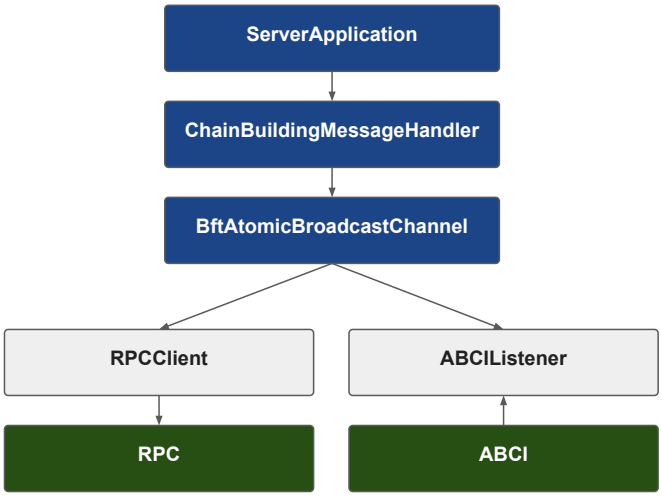

Figure 4.4: Launching Tendermint's interface

The code of the `BftAtomicBroadcastChannel` is given below and depicts the exact location in PROTECT, where the wrapper classes are instantiated. After an instance of the `BftAtomicBroadcastChannel` is created by the `ChainBuildingMessageHandler` first, the `link` method and then the `register` method is called and the `RPCClient` and `ABCIListener` are generated.

```
// BftAtomicBroadcastChannel.java
package com.ibm.pross.server.channel.bft;

import com.ibm.pross.server.channel.AtomicBroadcastChannel;
import com.ibm.pross.server.channel.ChannelListener;
import com.ibm.pross.server.channel.ChannelSender;

public class BftAtomicBroadcastChannel implements AtomicBroadcastChannel {
```

```
    private volatile ABCIListener wrapper;

    @Override
    public void register(final ChannelListener listener) throws InterruptedException
     {
        this.wrapper = new ABCIListener(listener);
    }

    public boolean isReady() {
        return ((this.wrapper != null) && (this.wrapper.isReady()));
    }

    @Override
    public ChannelSender link(final int senderId) {
        return new RPCClient(senderId);
    }

    @Override
    public void unregister(final ChannelListener listener) {
        throw new RuntimeException("not implemented");
    }
}
```

The wrapper classes itself perform the requirements specified in Sect. 4.2 and undertake the correct broadcast to Tendermint Core's RPC, as well as the appropriate request handling and processing from Tendermint Core's ABCI.

4.3.1 Broadcasting a message to Tendermint Core

The correct message broadcast by the `RPCClient` has to serialize a signed message, received from PROTECT's `ChainBuildingMessageHandler`, and then send it to Tendermint Core via RPC. Tendermint provides a REST interface [6] for broadcasting and stating queries which can be used by PROTECT's `RPCClient`. Inside its `broadcast` method, the transaction is appropriately constructed and then sent to Tendermint Core according to the mentioned REST interface specifications.

4.3.2 Receiving a message from Tendermint Core

The wrapper class `ABCIListener` must implement the ABCI in Java and communicate with Tendermint Core via a socket protocol. One possibility to implement this Java wrapper is the use of an existing Java implementation of the ABCI, the *jABCI* [5]. This open-source library allows for a straightforward integration into PROTECT's Java code-base as a *Maven* [1] dependency and comes with two example applications that demonstrate the use of the jABCI. Tendermint itself released a guide *Creating an application in Java* [9] recently, providing a step-by-step tutorial for building a simple distributed BFT key-value store in Java with implementing the ABCI. However, this guide uses *Gradle* [4] for building and managing the project dependencies. Since PROTECT is originally managed using Maven, the given manual cannot be adopted one-to-one for implementing the ABCI into PROTECT. In any case, PROTECT's `ABCIListener` has to create a socket that waits for Tendermint Core's requests and requires to implement the ABCI's methods to handle the requests appropriately.

The described system design and the implementation of the wrapper classes are still in development. The extensive documentation of the software components and the involved communication layers provided by this thesis, supports the necessary understanding for the ongoing implementation of Tendermint into PROTECT.

5 Conclusion and Future Work

This thesis tackled the upgrade of PROTECT's consensus layer. PROTECT is a software to build secure cryptographic services in realistic asynchronous networks, providing tunable fault-tolerance. While the theoretical and mathematical background of BFT, SMR and cryptography is known for a long time, approaches of their implementations and technology are still evolving. The main motivation for this thesis was the rise of Tendermint Core, a modular, language-agnostic and efficient consensus engine. Tendermint Core provides an opportunity to any deterministic state machine to perform the consensus in a scalable environment. Due to Tendermint's modular design, it is possible to implement to use it in existing software, using the provided interface, the ABCI. In order to use Tendermint Core for PROTECT, the existing consensus layer and it's interface were examined carefully. Currently, PROTECT's consensus layer is represented by BFT-SMaRT, a robust implementation of BFT SMR. This thesis documented not only the theoretical background and the properties of PROTECT, BFT-SMaRt and Tendermint, it further examined the details of the system architectures. In particular, the identification and examination of PROTECT's wrapper classes enabled the formulation of the required adaptions to use Tendermint Core. The intense study of Tendermint's documentation and testnet configuration provided further insight about Tendermint's functionalities, particularly in combination with Docker and Docker Compose.

This thesis represents an elaborate documentation about PROTECT's relevant system components concerning the interface to the consensus layer and the interactions among them. This knowledge will facilitate the implementation of the appropriate wrappers to use Tendermint's interface, which is a task for future work. Finally, it will be of interest to measure PROTECT's performance, by experimenting with varying network size.

A

DKG request in PROTECT

The example shown in Fig. A.1 illustrates how an incoming request for a DKG is handled by PROTECT. This request is handled by the `GenerateHandler` which initiates the DKG via the `ApvssShareholder` class. In the figure below the PROTECT instance, handling the incoming request on port 8081, is highlighted in yellow.

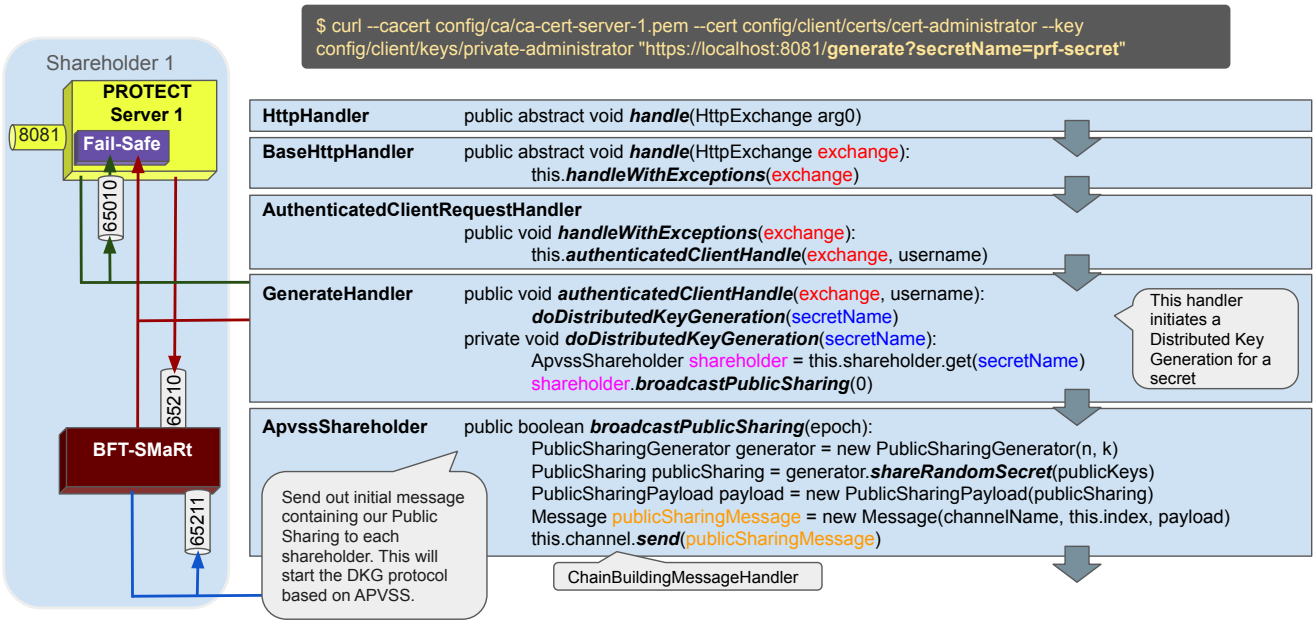

Figure A.1: Handling of a DKG request

The initial message for the secret sharing is signed by the `ChainBuildingMessageHandler` and serialized by the `BftChannelSender`, depicted in Fig. A.2. The message is finally sent to all instances of BFT-SMaRt's `ServiceProxy` by calling its method `invokeOrdered(serializedMessage)`.

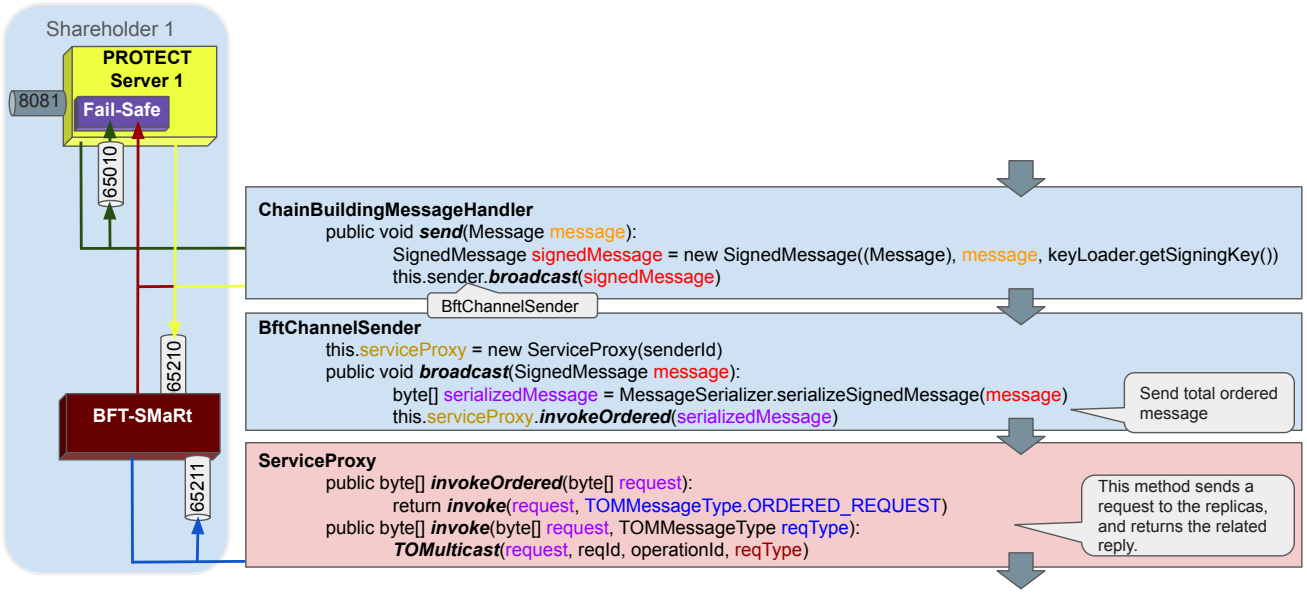

Figure A.2: Passing a message to BFT-SMaRt

PROTECT receives messages from BFT-SMaRt via its `ServiceReplica`. The `BftListenerWrapper` saves the received message and relays it to the `ChainBuildingMessageHandler`, which starts the extra validation procedure in the fail-safe layer, as described in Sec. 3.1.1.

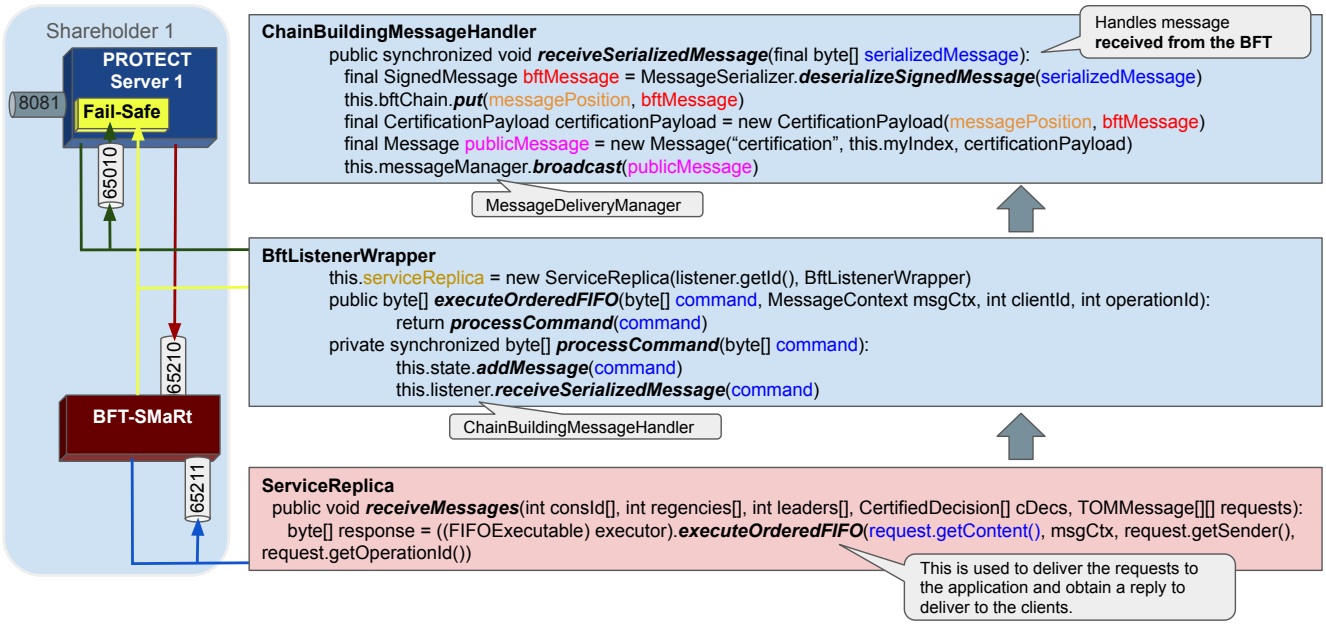

Figure A.3: Receiving a message from BFT-SMaRt

The `ChainBuildingMessageHandler` first deserializes, validates and adds the message to the BFT log. Then the broadcast of the signed and certified message and its position in the BFT log is performed with the help of the `MessageDeliveryManager`, as illustrated in Fig. A.4.

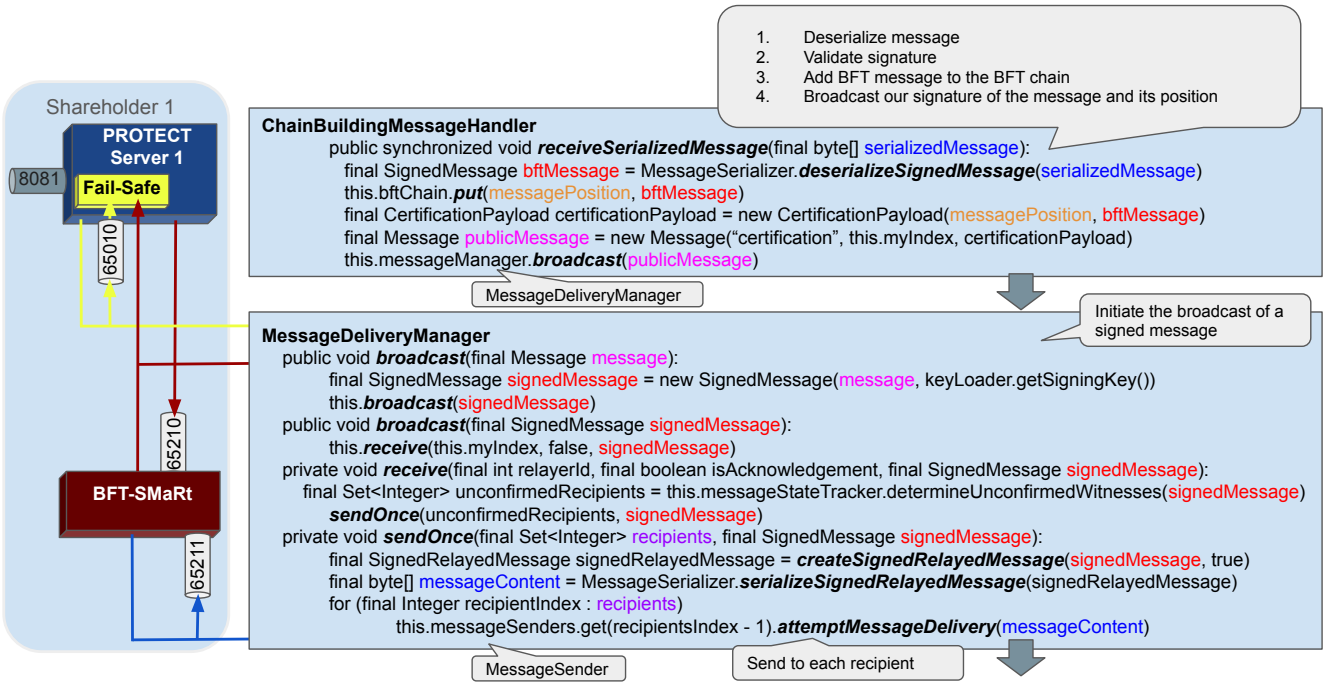

Figure A.4: Preparing the protocol in the fail-safe layer

B
Testnet with Docker

This docker-compose.yml file is provided by Tendermint [8] and can be used to launch a four-node-testnet with an in-process example kvstore-application. Furthermore, this file is can be used to add and configure more nodes in the network. Finally, this file is used to configure a network with an own application.

```
version: '3'

services:
  node0:
    container_name: node0
    image: "tendermint/localnode"
    ports:
      - "26656-26657:26656-26657"
    environment:
      - ID=0
      - LOG=${LOG:-tendermint.log}
    volumes:
      - ./build:/tendermint:Z
    networks:
      localnet:
        ipv4_address: 192.167.10.2

  node1:
    container_name: node1
    image: "tendermint/localnode"
    ports:
      - "26659-26660:26656-26657"
    environment:
      - ID=1
      - LOG=${LOG:-tendermint.log}
    volumes:
      - ./build:/tendermint:Z
    networks:
      localnet:
        ipv4_address: 192.167.10.3

  node2:
    container_name: node2
    image: "tendermint/localnode"
    environment:
      - ID=2
      - LOG=${LOG:-tendermint.log}
```

```
    ports:
      - "26661-26662:26656-26657"
    volumes:
      - ./build:/tendermint:Z
    networks:
      localnet:
        ipv4_address: 192.167.10.4

  node3:
    container_name: node3
    image: "tendermint/localnode"
    environment:
      - ID=3
      - LOG=${LOG:-tendermint.log}
    ports:
      - "26663-26664:26656-26657"
    volumes:
      - ./build:/tendermint:Z
    networks:
      localnet:
        ipv4_address: 192.167.10.5

networks:
  localnet:
    driver: bridge
    ipam:
      driver: default
      config:
      -
        subnet: 192.167.10.0/16
```

This part of Tendermint's Makefile is used to launch a testnet with a single command. Note that the command `build-docker-localnode` executes another Makefile, that is given below.

```
# Local testnet using docker

# Build linux binary on other platforms
build-linux: tools
   GOOS=linux GOARCH=amd64 $(MAKE) build

build-docker-localnode:
   @cd networks/local && make

# Runs `make build_c` from within an Amazon Linux (v2)-based Docker build
# container in order to build an Amazon Linux-compatible binary. Produces a
# compatible binary at ./build/tendermint
build_c-amazonlinux:
   $(MAKE) -C ./DOCKER build_amazonlinux_buildimage
   docker run --rm -it -v `pwd`:/tendermint tendermint/tendermint:build_c-
    amazonlinux

# Run a 4-node testnet locally
localnet-start: localnet-stop build-docker-localnode
   @if ! [ -f build/node0/config/genesis.json ]; then docker run --rm -v $(CURDIR)/
    build:/tendermint:Z tendermint/localnode testnet --config /etc/tendermint/config-
    template.toml --v 4 --o . --populate-persistent-peers --starting-ip-address
    192.167.10.2; fi
   docker-compose up

# Stop testnet
localnet-stop:
   docker-compose down
```

This Makefile triggers the building of the Docker image for the Tendermint nodes, using the Dockerfile given in below.

```
# Makefile for the "localnode" docker image.
all:
   docker build --tag tendermint/localnode localnode
.PHONY: all
```

This Dockerfile is used by Tendermint to build the Docker image for the Tendermint nodes.

```
FROM alpine:3.7
MAINTAINER Greg Szabo <greg@tendermint.com>

RUN apk update && \
    apk upgrade && \
    apk --no-cache add curl jq file

VOLUME [ /tendermint ]
WORKDIR /tendermint
EXPOSE 26656 26657
ENTRYPOINT ["/usr/bin/wrapper.sh"]
CMD ["node", "--proxy_app", "kvstore"]
STOPSIGNAL SIGTERM

COPY wrapper.sh /usr/bin/wrapper.sh
COPY config-template.toml /etc/tendermint/config-template.toml
```

Bibliography

[1] "Apache Maven." https://maven.apache.org/.

[2] "Cosmos SDK." https://tendermint.com/sdk/.

[3] "Docker." https://www.docker.com/.

[4] "Gradle Build Tool." https://gradle.org/.

[5] "jABCI." https://github.com/jTendermint/jabci.

[6] "RPC client for Tendermint." https://docs.tendermint.com/master/rpc/.

[7] "Tendermint." https://tendermint.com/.

[8] "Tendermint code." https://github.com/tendermint/tendermint.

[9] "Tendermint core documentation." https://docs.tendermint.com/.

[10] "Tunable protocols for threshold and proactive cryptography.".

[11] J. Andress, *The basics of information security: understanding the fundamentals of InfoSec in theory and practice*. Syngress, 2014.

[12] A. Bessani, J. Sousa, and E. E. Alchieri, "State machine replication for the masses with bft-smart," in *2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pp. 355–362, IEEE, 2014.

[13] R. E. Blahut, *Cryptography and secure communication*. Cambridge University Press, 2014.

[14] C. Cachin, "Distributed cryptography and proactive security." https://cachin.com/cc/sft13/.

[15] C. Cachin, R. Guerraoui, and L. Rodrigues, *Introduction to reliable and secure distributed programming*. Springer Science & Business Media, 2011.

[16] R. Canetti, R. Gennaro, A. Herzberg, and D. Naor, "Proactive security: Long-term protection against break-ins," *RSA Laboratories CryptoBytes*, vol. 3, no. 1, pp. 1–8, 1997.

[17] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Transactions on Computer Systems (TOCS)*, vol. 20, no. 4, pp. 398–461, 2002.

[18] A. Clement, M. Kapritsos, S. Lee, Y. Wang, L. Alvisi, M. Dahlin, and T. Riche, "Upright cluster services," in *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pp. 277–290, ACM, 2009.

[19] W. Diffie and M. E. Hellman, "Multiuser cryptographic techniques," in *Proceedings of the June 7-10, 1976, national computer conference and exposition*, pp. 109–112, ACM, 1976.

[20] D. Evans, V. Kolesnikov, M. Rosulek, *et al.*, "A pragmatic introduction to secure multi-party computation," *Foundations and Trends® in Privacy and Security*, vol. 2, no. 2-3, pp. 70–246, 2018.

[21] R. Gennaro, S. Jarecki, H. Krawczyk, and T. Rabin, "Secure distributed key generation for discrete-log based cryptosystems," *Journal of Cryptology*, vol. 20, no. 1, pp. 51–83, 2007.

[22] A. Herzberg, S. Jarecki, H. Krawczyk, and M. Yung, "Proactive secret sharing or: How to cope with perpetual leakage," in *Annual International Cryptology Conference*, pp. 339–352, Springer, 1995.

[23] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," in *Concurrency: the Works of Leslie Lamport*, pp. 203–226, 2019.

[24] National Institute of Standards and Technology, "Threshold cryptography." `https://csrc.nist.gov/projects/threshold-cryptography`.

[25] M. Pease, R. Shostak, and L. Lamport, "Reaching agreement in the presence of faults," *Journal of the ACM (JACM)*, vol. 27, no. 2, pp. 228–234, 1980.

[26] J. Resch, "Platform for robust threshold cryptography." `https://csrc.nist.gov/Presentations/2019/Platform-for-Robust-Threshold-Cryptography`.

[27] J. Resch, "Protect." `https://github.com/jasonkresch/protect`.

[28] R. L. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems," *Communications of the ACM*, vol. 21, no. 2, pp. 120–126, 1978.

[29] F. B. Schneider, "Implementing fault-tolerant services using the state machine approach: A tutorial," *ACM Computing Surveys (CSUR)*, vol. 22, no. 4, pp. 299–319, 1990.

[30] A. Shamir, "How to share a secret," *Communications of the ACM*, vol. 22, no. 11, pp. 612–613, 1979.

[31] University of Lisbon, LaSIGE research unit, "BFT-SMaRt." `https://github.com/bft-smart/library`.

Acknowledgments

I would like to thank Prof. Dr. Christian Cachin for the supervision of this thesis and for countless inspirational discussions. I'm highly thankful for the professional support and the great work atmosphere provided by him and his research team. Despite a rather busy semester, the team and myself have enjoyed the outmost attention by our supervisor, for which I'm very grateful for.

In particular, I further like to thank the Cryptology and Data Security Group for being good friends and motivated and helpful scientists that never hesitated to support me in any situation. A further appreciation goes to Bettina Choffat for her assistance in all administrative matter.

A big thank you to all my friends and family who supported me during the master thesis.

Erklärung

gemäss Art. 30 RSL Phil.-nat. 18

Name/Vorname: Froidevaux Nathalie

Matrikelnummer: 12-124-590

Studiengang: Computer Science

Bachelor ☐ Master ☑ Dissertation ☐

Titel der Arbeit: Threshold Cryptography with Tendermint Core

LeiterIn der Arbeit: Prof. Dr. Christian Cachin

Ich erkläre hiermit, dass ich diese Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen benutzt habe. Alle Stellen, die wörtlich oder sinngemäss aus Quellen entnommen wurden, habe ich als solche gekennzeichnet. Mir ist bekannt, dass andernfalls der Senat gemäss Artikel 36 Absatz 1 Buchstabe r des Gesetzes vom 5. September 1996 über die Universität zum Entzug des auf Grund dieser Arbeit verliehenen Titels berechtigt ist.
Für die Zwecke der Begutachtung und der Überprüfung der Einhaltung der Selbständigkeitserklärung bzw. der Reglemente betreffend Plagiate erteile ich der Universität Bern das Recht, die dazu erforderlichen Personendaten zu bearbeiten und Nutzungshandlungen vorzunehmen, insbesondere die schriftliche Arbeit zu vervielfältigen und dauerhaft in einer Datenbank zu speichern sowie diese zur Überprüfung von Arbeiten Dritter zu verwenden oder hierzu zur Verfügung zu stellen.

Ort/Datum Bern 27.01.2020

Unterschrift