Department of Informatics
Technical University of Munich

TECHNICAL UNIVERSITY OF MUNICH

DEPARTMENT OF INFORMATICS

MASTER'S THESIS IN INFORMATICS

Trustworthy Configuration Management with Distributed Ledgers

Valentin Johannes Hauner

Technical University of Munich

Department of Informatics

Master's Thesis in Informatics

Trustworthy Configuration Management with Distributed Ledgers

Vertrauenswürdiges Konfigurationsmanagement mit verteilten Ledgers

Author:	Valentin Johannes Hauner
Supervisor:	Prof. Dr.-Ing. Georg Carle
Advisor:	Dr. rer. nat. Holger Kinkelin
	Dr. rer. nat. Heiko Niedermayer
Date:	May 15, 2018

I confirm that this Master's Thesis is my own work and I have documented all sources and material used.

Garching, May 15, 2018

Location, Date

Signature

Abstract

In recent years, IT systems have been adopted in a wide range of uses with direct consequences for the physical world, including the Internet of Things (IoT), health care and law enforcement. The rising complexity of these systems requires a proper concept for Configuration Management (CM) to guarantee their security and safety. Incidents in the recent past show that many organizations have to cope with attacks from outside, but also with failures and abuse of trust from inside: an individual with administrative privileges typically has the ability to change critical parts of a system's configuration, without requiring the approval of additional administrators.

To mitigate attacks involving the abuse of administrative privileges, we present **TANCS**, a **T**amper-Resistant and **A**uditable **N**etwork **C**onfiguration **S**ystem. It introduces the principle of Multi-Party Authorization (MPA), requiring multiple parties to review and approve each critical configuration change. Only after the configuration change has been authorized by multiple parties stipulated in a policy, it can be applied to the targeted device. For every configuration management operation performed with TANCS, we ensure accountability and traceability. Furthermore, TANCS is designed to be tamper-resistant, preventing attempts by individuals with administrative privileges to attack or circumvent the configuration management process itself.

For providing tamper-resistance, we leverage Distributed Ledger Technology (DLT). In particular, TANCS is based on Hyperledger Fabric, a comprehensive framework developed by the Linux Foundation for operating a distributed ledger together with a distributed execution environment. Every configuration management operation is executed as a smart contract in a peer-to-peer network, with the execution results stored redundantly in the ledger copies of the peers. That way, no single participant of the network is able to forge or erase the outcomes of an operation. Additionally, it even becomes possible to manage configurations for IT infrastructure and services shared across different stakeholders which only share a limited amount of trust, like different organizational units or a consortium of several organizations.

Zusammenfassung

In letzter Zeit wurden IT-Systeme in einer Vielzahl von Einsatzbereichen mit unmittelbaren Auswirkungen auf die reale Welt eingeführt, vom Internet der Dinge über das Gesundheitswesen bis hin zur Strafverfolgung. Die steigende Komplexität dieser Systeme erfordert ein geeignetes Konzept zur Konfigurationsverwaltung, um deren Sicherheit und Gefahrlosigkeit zu gewährleisten. Vorfälle aus der jüngsten Vergangenheit offenbaren, dass viele Organisationen mit Angriffen von außerhalb, aber auch mit Versagen und Vertrauensmissbrauch aus dem Inneren zu kämpfen haben: Ein Einzelner mit administrativen Berechtigungen ist üblicherweise dazu in der Lage, kritische Bestandteile einer Systemkonfiguration zu verändern, ohne auf die Zustimmung anderer Administratoren angewiesen zu sein.

Um Angriffe auf Basis des Missbrauchs administrativer Berechtigungen zu entschärfen, stellen wir **TANCS** vor, ein **T**amper-Resistant and **A**uditable **N**etwork **C**onfiguration **S**ystem. Es führt das Prinzip der Mehrparteienautorisierung ein, das die Überprüfung und Zustimmung mehrerer Parteien für jede kritische Konfigurationsänderung erfordert. Erst nachdem die Konfigurationsänderung von mehreren, in einer Policy festgelegten Parteien autorisiert worden ist, kann sie auf dem Zielgerät angewendet werden. Für jeden mit TANCS ausgeführten Konfigurationsverwaltungsvorgang stellen wir Verantwortlichkeit und Rückverfolgbarkeit sicher. Darüber hinaus ist TANCS auf Manipulationssicherheit ausgelegt, sodass die Versuche Einzelner mit administrativen Berechtigungen, den Konfigurationsverwaltungsprozess anzugreifen oder zu umgehen, verhindert werden.

Um diese Manipulationssicherheit zu erbringen, machen wir uns Distributed Ledger Technology zunutze. TANCS basiert im Speziellen auf Hyperledger Fabric, ein umfangreiches Framework der Linux Foundation, um einen verteilten Ledger zusammen mit einer verteilten Ausführungsumgebung zu betreiben. Jeder Konfigurationsverwaltungsvorgang wird als Smart Contract in einem Peer-to-Peer-Netzwerk ausgeführt, wobei die Berechnungsergebnisse redundant in den Ledger-Kopien der Peers gespeichert werden. Auf diese Weise ist kein einzelner Beteiligter in der Lage, die Ergebnisse eines Vorgangs zu fälschen oder zu löschen. Außerdem wird die Möglichkeit geschaffen, Konfigurationen für gemeinschaftliche IT-Infrastruktur und -Dienstleistungen zu verwalten, die von verschiedenen Stakeholdern mit einem beschränkten Maß an Vertrauen untereinander betrieben werden, wie zum Beispiel verschiedene Organisationseinheiten oder ein Konsortium mehrerer Organisationen.

Acknowledgements

The essence of all beautiful ~~art~~ science, all great ~~art~~ science, is gratitude.
– *Friedrich Wilhelm Nietzsche*

First of all, I would like to thank Dr. rer. nat. Holger Kinkelin and Dr. rer. nat. Heiko Niedermayer for providing me with important advice and expert knowledge during this research project. I have learned a lot.

It was them who encouraged me to participate in writing a scientific publication about this project, which was a great experience with a steep learning curve for myself.

In addition, I would like to thank Prof. Dr.-Ing. Georg Carle for supervising me. He and his research associates at the Chair of Network Architectures and Services provided me with a valuable insight into the scientific world and community.

Publications

Holger Kinkelin, Valentin Hauner, Heiko Niedermayer, and Georg Carle. "Trustworthy Configuration Management for Networked Devices using Distributed Ledgers". In: *NOMS 2018 – IEEE/IFIP DOMINOS Workshop*. Taipei, Taiwan, Apr. 2018

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Our work discusses reasons and consequences of IT attacks caused by insiders like fraudulent administrators and introduces a novel approach for managing configurations in a trustworthy manner based on Distributed Ledger Technology (DLT). In the following, we explain the motivation behind our work and state its goals, followed by a short outline of the thesis' structure.

1.1 Motivation

Nowadays, IT systems become more and more complex. This is not only caused by an increasing amount of features, but also by a growing interconnection of multiple systems, leading to new challenges for managing, configuring and securing them. Elaborate toolsets and frameworks are necessary to cope with this administrative effort. They try to reduce complexity and assist administrators in maintaining the system landscape, implementing security guidelines and reacting to imminent threats.

However, a well-configured and secure system should not only prevent and fight attacks on the software and infrastructure itself, but also those that involve the abuse of administrative privileges. Advanced Persistent Threats (APTs) describe individual and precisely prepared attacks targeted to a specific entity [12, p. 3], like the administrator of an organization. After the entity has been compromised, the attackers are able to abuse the administrative privileges, for instance by stealing sensitive information or by causing severe damage to the system landscape.

Apart from that, administrators can make mistakes by accident. Furthermore, they can go rogue and harm their employer deliberately. In both cases, the misuse of their

administrative privileges is likely to cause severe damage to the IT systems they are responsible for. Logging and auditing systems provide means to detect anomalies of such kind, but typically in hindsight only. Moreover, administrative privileges often allow to bypass or disable common security measures, like access control, an Intrusion Detection System (IDS) or solutions for Security Information and Event Management (SIEM).

This is where our approach for trustworthy configuration management comes into play. We aim to mitigate and avoid the attacks and incidents described above with a configuration management system that establishes control over the actions of administrators and effectively restrains their privileges.

1.2 Goals

The goal of this work is to design and implement a configuration management system that enforces a configuration change to be explicitly reviewed and approved by independent auditors. For that purpose, we conduct a configuration management process requiring Multi-Party Authorization (MPA) to achieve Byzantine Fault Tolerance (BFT) against attacks and failures caused by an administrator's configuration change. Only after a configuration change has been authorized by a set of experts, the targeted devices retrieve it from our system and apply it locally. We call our system **TANCS**, referring to a **T**amper-Resistant and **A**uditable **N**etwork **C**onfiguration **S**ystem.

The different parties that need to authorize a configuration change can be specified on a per-device basis, making it possible to take into account the criticality of a device. For the whole configuration management process, we guarantee accountability and traceability: every management operation performed with our system is cryptographically associated with an actor and recorded in an append-only log. As our system is designed to be tamper-resistant, it even prevents entities with administrative privileges from manipulating or negatively affecting its service.

To ensure that administrators cannot continue to access and configure the target devices directly, the devices have to be locked appropriately. This involves the shutdown of network services that allow remote logins, such as SSH, and the reduction of system accounts with administrative privileges to a minimum. Depending on the criticality of the device, however, it may still be necessary to be able to administrate it directly in cases of emergency.

The technical basis for our system is provided by Distributed Ledger Technology (DLT). Thereby, we are able to conduct the configuration management process in a decentralized and unforgeable manner. Additionally, DLT offers the ability to manage configurations for infrastructure shared across different stakeholders which only share a limited amount of trust. A typical example includes two or more organizations that want to run and administer an IT service collectively, with each of them contributing own infrastructure elements and application instances to it.

1.3 Outline

The thesis is structured as follows: chapter 2 introduces background information on the subjects of Configuration Management (CM), Multi-Party Authorization (MPA) and Distributed Ledger Technology (DLT). In chapter 3, we analyze the basic problem, formulate requirements for our system and propose different approaches to choose from, before chapter 4 presents existing work related to our topic. Our system's high-level design is explained in chapter 5, followed by the implementation in chapter 6. We evaluate our system with respect to interoperability, performance and security in chapter 7, before we conclude in chapter 8.

Chapter 2

Background

This chapter introduces the fundamentals needed for understanding our work. It covers the concept of Configuration Management, an authorization model for multiple parties, and the characteristics and implementations of Distributed Ledger Technology.

2.1 Configuration Management

Configuration Management (CM) is a broadly defined concept for establishing, organizing and maintaining the configuration of an artifact throughout its whole lifecycle. Although the term is used in a multitude of sectors, including civil engineering and military services, we only consider configuration management for software, also known as Software Configuration Management (SCM). This discipline is highly relevant for our work, since it deals with configurations for applications, like a web server, as well as with configurations relevant to infrastructure components, such as firewall rules.

2.1.1 Concept

Managing configurations for software is a complex and critical task. It includes not only the management of settings and operational modes of a software, but also its introduction, installation and replacement. As soon as the number of artifacts to manage increases, the establishment of a configuration management process becomes inevitable. Only with such a well-defined process, consistency, continuity and changeability can be guaranteed for a software and its configuration.

2.1.2 Challenges

Configuration Management (CM) comes along with several challenges as mentioned by Morris [52, chap. 1], with the most important ones for our work described below.

Consistency

The configuration of infrastructure components providing the same service should be consistent. A typical example is a group of server instances behind a load balancer. Every server instance should run the same version of the server software together with the same configuration. However, some parts of the configuration may differ, like those needed for identifying an instance.

Consistency is essential for avoiding configuration drifts, a pattern that describes components which have initially been set up with the same configuration, but got configured individually over time. Furthermore, consistency is an important prerequisite for automation. Only if each component of a service is configured consistently, it will be convenient and safe to employ an automated process that adapts their configuration and sets up new components.

Continuity

A service provided by infrastructure components should be continuously available. No configuration change should lead to a disruption of the service. The continuity principle usually includes Disaster Recovery (DR), a process that is triggered in case of incidents beyond the administrators' control, like an unexpected hardware failure, and takes measures to guarantee the service's continuity.

Changeability

It should be possible to change the configuration of a service and its components at any time with as minimal effort as possible. Therefore, applying configuration changes should be a routine task that is well-practised and guided by a configuration management framework. Applying small changes one by one should be preferred over applying a huge set of changes at one go.

Testability

It should be possible to easily test a configuration change before applying it to the production environment. Consequently, configuration changes should be applied to a

dedicated testing environment beforehand. The testing process may be supported by the configuration management framework.

Version Control

All configuration changes should be put under version control in order to understand who configured what in which way. This information may also be used for conducting an auditing process. Besides, it simplifies the reproducibility of misconfigurations and their rollback.

Documentation

All configuration changes should be documented. In the simplest case, the documentation is written and maintained manually. Since documentations can become outdated quickly, configuration management frameworks should provide means to easily manage and adapt the documentation.

2.1.3 Approaches

A configuration management process can be implemented in different ways. For small environments, it may suffice to conduct it manually. In contrast, interactive approaches provide a configuration management process guided and executed by dedicated management tools, while Infrastructure as Code (IaC) is a paradigm to automatically configure software via human- and machine-readable configuration scripts.

Manual Approaches

A configuration management process may be conducted manually. However, this approach is feasible for small environments only, like a landscape consisting of a few standalone servers. It is still possible to meet every challenge described in section 2.1.2, but the manual operation requires a high degree of coordination within the administration team.

Interactive Approaches

With an interactive approach, the configuration management process is guided and executed by a dedicated configuration management framework. It provides a unified user interface for the administration team to manage the configurations of a multitude of services and components. The framework assists the user in meeting the challenges

described in section 2.1.2 and may offer additional features, like advanced access control and monitoring.

Infrastructure as Code

Infrastructure as Code (IaC) is a paradigm to configure a dynamic infrastructure environment via human- and machine-readable configuration scripts [52, p. 5]. In contrast to other approaches, it allows for a high degree of automation. IaC is suitable for managing large environments with little effort and therefore solves the scaling problems for configuration management that came along with the concepts of virtualization and Infrastructure as a Service (IaaS).

Concept: With IaC, configuration scripts are applied to a set of targets. The scripting language differs depending on the framework and supports an imperative or declarative style or a mixture of both. In contrast to imperative scripts, declarative scripts describe what configuration has to be applied and not how it has to be applied. Consequently, idempotence can be achieved easily with a declarative approach, i.e. applying the same script more than once to a target leads to the same result.

Since several of these scripting languages are optimized to be human-readable, the configuration scripts are usually self-documenting. No extensive documentation that can become outdated is needed. Furthermore, even users without a deep technical knowledge are able to understand the definition file.

An IaC framework can be operated in two different modes: *push* and *pull*. While configurations are delivered to the targets by the management framework in the first case, targets retrieve configurations from the management framework on their own in the latter case. Depending on the framework and the operating mode, it may be necessary to install additional software on the targets that handles the communication with the management framework.

Continuous Configuration Automation (CCA) is a concept based on IaC and aims to automate the tasks performed with IaC frameworks [24], similar to the goals that are pursued with Continuous Integration (CI) and Continuous Delivery (CD) in the development and operations sector.

Frameworks: A wide variety of IaC frameworks is available, with the most popular ones being Chef, Puppet and Ansible. Its intended use cases, scripting languages and operation modes differ, which makes it necessary to carefully evaluate which one fits best

for the environment to manage. For that purpose, Delaet et al. provide an extensive comparison framework that proposes a multitude of evaluation properties to consider when adopting a configuration management tool [20].

Chef Chef is a widely-used IaC framework implemented in Ruby and Erlang and was initially released in 2009 [17]. It uses a Domain-Specific Language (DSL) based on Ruby for its configuration scripts. A single configuration is expressed as a *recipe*, with multiple recipes combined into a *cookbook*. The DSL supports anything that can be done with Ruby and therefore can be seen as a full-featured programming language. Nonetheless, the language can be used in a declarative way: different *resources* that come along with different *actions* can be used to define a desired state of a target, as shown in figure 2.1. Chef has to be installed on both a server and a client and follows the *pull* mechanism. The software comprises a large toolset together with community features, including a testing suite and a public repository with thousands of predefined cookbooks for well-known software.

```
1 httpd_service 'default' do
2   action [:create, :start]
3 end
4
5 httpd_config 'default' do
6   source 'mysite.cnf.erb'
7   notifies :restart, 'httpd_service[default]'
8   action :create
9 end
```

Figure 2.1: Example of a set of Chef recipes [16], with the first one creating and starting a new *httpd* instance and the second one configuring it with a Ruby-based configuration file template

Puppet Puppet, with its initial release in 2005, brings its own declarative configuration language inspired by the Nagios configuration file format [60]. Puppet follows the client-server architecture, with the client denoted by *agent* and the server denoted by *master*. The client fetches configurations from the server and therefore conducts a *pull* mechanism. Language files are called *manifests* and contain a set of resource declarations with a type, a title and attributes. The latter describe the desired state of the resource. Figure 2.2 shows an exemplary manifest with several resource declarations.

Ansible Ansible is an IaC framework written in Python and was initially released in 2012 [3]. Configurations are declared in *playbooks*. Each playbook may comprise several *plays*, which in turn may include several *tasks*. Playbooks are not expressed using a

```
1  package { 'ntp':
2    ensure => installed,
3  }
4
5  service { 'ntp':
6    name      => 'ntp',
7    ensure    => running,
8    enable    => true,
9    subscribe => File['ntp.conf'],
10 }
11
12 file { 'ntp.conf':
13   path    => '/etc/ntp.conf',
14   ensure  => file,
15   require => Package['ntp'],
16   source  => "puppet:///modules/ntp/ntp.conf",
17 }
```

Figure 2.2: Example of a Puppet manifest [58], containing resource declarations of the types `package`, `service` and `file`, with the first one ensuring that the *ntp* package is installed, the second one ensuring that the *ntp* service is running and the third one configuring it with a file stored on the Puppet *master*

script or programming language, but with declarations in the YAML syntax. That way, playbooks remain simple to read, even for non-programmers, and can easily be designed to support idempotence. Each play is targeted to a set of *hosts* which are managed in an *inventory* file that contains their IP addresses and domain names. Furthermore, it is possible to define groups of hosts. Figure 2.3 shows an exemplary playbook with a single play and several tasks targeted to a host group.

The main difference between Ansible and other IaC frameworks is its agentless architecture: no additional software needs to be installed on the targets to apply configurations to, referred to as *managed nodes* in the Ansible terminology. Consequently, Ansible follows the *push* mechanism: a *control machine* runs a playbook by directly accessing the managed nodes via SSH. Additionally, the framework supports a *pull* mechanism where managed nodes retrieve playbooks from a version-controlled repository and run them locally.

2.2 Multi-Party Authorization

Multi-Party Authorization (MPA) is a security model based on redundant supervision. It requires multiple parties to authorize a certain action. Without the authorization of multiple parties, the action is not allowed to be carried out. Before parties authorize an action, it is their responsibility to carefully review and assess it. The requirements for an action to become finally authorized can be formulated in a policy or negotiated dynamically. Typical requirements comprise a minimum number of authorizers, a set

```
---
- hosts: webservers
  vars:
    http_port: 80
    max_clients: 200
  remote_user: root
  tasks:
  - name: ensure apache is at the latest version
    yum: name=httpd state=latest
  - name: write the apache config file
    template: src=/srv/httpd.j2 dest=/etc/httpd.conf
    notify:
    - restart apache
  - name: ensure apache is running (and enable it at boot)
    service: name=httpd state=started enabled=yes
  handlers:
    - name: restart apache
      service: name=httpd state=restarted
```

Figure 2.3: Example of an Ansible playbook [6], with a single play targeted to the host group *webservers*, and *httpd*-related tasks to check its version, to write a template-based configuration file with a subsequent restart, and to finally check if it is running

of specific authorizers, role-based properties or any combination of these criteria, like requiring the authorization from the department manager and at least k administrators for a major software upgrade.

MPA can be used to protect critical systems from attacks and mistakes of individuals, like administrative environments and storage systems for critical data. Depending on the particular use case, it may be desirable to ensure accountability and traceability for the whole authorization process, for instance by employing a logging and auditing system.

Lin introduces a formal definition of MPA in her PhD thesis [35, sec. 2.3]. According to that, the n-party authorization problem is a tuple defined as follows:

$$\begin{aligned} &(e_0, m_1, ..., m_n, A, M, Z, S, \prec) \\ &A, M, Z \subseteq N = \{1, ..., n\} \\ &\prec : N \to N \end{aligned} \tag{2.1}$$

e_0 is the entity that initiates the authorization and $m_1, ..., m_n$ are n different names of entities that are known to e_0. A represents the set of authorizers, i.e. the entities that grant the authorization. M represents the set of resource managers, i.e. the entities that enforce the authorization, like a document management system. Z represents the set of authorizees, i.e. the entities that is granted the authorization, like a printing service. S

denotes an authorization scope, i.e. a set of resources and an operation defined on them, like reading files. $\prec$ defines the order of the authorizations to be granted by entities, i.e. $i \prec j$ means that e_i should grant the authorization before e_j. The meta information of the actual authorization request is described by $(m_1, ..., m_n, A, M, Z, S, \prec)$.

For a protocol to solve the n-party authorization problem, Lin defines the following four properties to be met:

1. Agreement: honest entities agree on the authorization request's content
2. Validity: honest entities process the authorization request initiated by e_0, without any modifications
3. Temporal order: authorizations are granted in the order defined by $\prec$
4. Non-triviality: the protocol is not allowed to simply abort

A simple example for an authorization request looks as follows: the initiator, e_0, is a printing service. m_1 is the name of the printing service (which also corresponds to e_1), m_2 is the name of a document management system (e_2), and m_3 is the name Alice (e_3). Alice is the authorizer, therefore $A = \{3\}$. The document management system is the resource manager, therefore $M = \{2\}$. The printing service is the authorizee, therefore $Z = \{1\}$. The authorization scope S deals with reading documents. Finally, the order is defined as $3 \prec 2$ and $2 \prec 1$. Consequently, Alice should grant the authorization before the document management system, followed by the printing service.

2.3 Distributed Ledger Technology

Distributed Ledger Technology (DLT) is a technology for managing a ledger in a peer-to-peer network and without the need for a central authority. It became increasingly popular after Nakamoto had published the concept of a DLT-based electronic cash system called *Bitcoin* in 2008 [53], which utilizes a *blockchain* as data structure and Proof of Work (PoW) as consensus mechanism. From that time on, plenty of research has been conducted in this field, exploring not only use cases for electronic cash systems, but also covering generalized approaches, like the distributed execution of smart contracts.

2.3.1 Concept

A distributed ledger can be seen as a decentralized database with a global state, as stated by Natarajan et al. in their report for the World Bank [54, chap. 1]. Although,

in its original definition, a ledger is a kind of journal that permanently keeps track of debits, credits and the resulting balance of a customer's account, a digital ledger is basically able to cope with any kind of data record. These data records are appended in a sequential order and in an unerasable and unforgeable way. The ledger is collectively maintained by multiple peer nodes that need to reach a consensus on its state. Neither a central authority nor any other trusted third party is involved in the management of the ledger. Furthermore, DLT aims to prevent a collusion between possibly malicious peers from effectively manipulating the ledger.

All peer nodes store an identical copy of the ledger and communicate via a broadcast model that is usually implemented in a best-effort manner, like gossip. A participant of the network can initiate a new transaction that eventually will lead to a modification of the ledger and its state. Only valid transactions can lead to a modification of the ledger's state. Common validity checks include, for instance, the double-spending problem, i.e. that no asset is spent more than once by its owner. The consensus algorithm decides which transactions are appended to the ledger in which order. Its implementation differs depending on the specific distributed ledger system. Typical implementations are Proof of Work (PoW), Proof of Stake (PoS), and consensus models based on Byzantine Fault Tolerance (BFT).

2.3.2 Approaches

A distributed ledger can be operated and implemented in different ways. A major differentiation is made between *unpermissioned* and *permissioned* ledgers: while anyone can join the corresponding peer-to-peer network and participate in the management of the ledger in the first case, permissioned ledgers restrict the participation to a specific set of entities. A typical example for an unpermissioned ledger is the Bitcoin payment system, whereas so-called corporate ledgers between two or more organizations used for transferring business assets and negotiating contracts belong to the domain of permissioned ledgers. Additionally, *public* ledgers are distinguished from *private* ledgers: the first type grants read access to everyone, even if the ledger is permissioned, but the second type prevents unauthorized entities from accessing the ledger in any case, even for read attempts.

Currently, there are two major ways of implementing a distributed ledger: with a *blockchain* or with a Directed Acyclic Graph (DAG). At the time of writing this thesis, most of the ledger systems are using a blockchain. As the name implies, a blockchain is a continuously growing chain of blocks. Each block comprises a set of transactions and a cryptographically secure identifier of the previous block in the chain, usually a

hash value. That way, all blocks are connected to a chain in an unforgeable manner. For a blockchain-based distributed ledger, the consensus algorithm running on the peer nodes typically decides which peer is allowed to append the next block containing a set of new transactions to the chain. Until now, the most prominent approach is PoW, which is based on a time- and resource-consuming competition to solve a cryptographic puzzle by trying to find a hash value with certain properties. The winner of the competition is then allowed to append a new block and usually receives a reward for the work. However, there are prospects of more efficient consensus algorithms, like PoS.

A different approach for implementing a distributed ledger involves using a DAG. The cryptocurrency system *IOTA* designed for the Internet of Things (IoT) and proposed by Popov in 2017 [57] is a prominent example in this field. Popov introduces a *tangle* graph, with its vertices representing transactions and its edges representing approvals. Each new transaction must approve 2 (or in general k) other transactions. The main idea of this approach is to require an issuer of a new transaction to approve other valid transactions and therefore contribute to the network's continuity and security. According to Popov, advantages of this approach over a blockchain are a significant reduction of latency, resource consumption and transaction fees, making the system suitable for micropayments.

2.3.3 Implementations

Due to the success of the Bitcoin payment system, a considerable number of DLT implementations has emerged in recent years. In the following, we introduce the most prevalent ones.

Bitcoin

The Bitcoin payment system is based on a public and unpermissioned distributed ledger that is implemented with a blockchain [53]. The consensus on new blocks is reached via PoW. Each block, limited in its size, contains a set of transactions. Each transaction describes a flow of Bitcoin units from at least one sender address, the so-called input, to at least one receiver address, the so-called output. The inputs of a transaction must be unspent and at least as great as the outputs. Each participant owns a wallet that includes at least one pair of a public and private key. A public key is used for the calculation of a Bitcoin address, while a private key is used to sign a transaction.

After the participant has signed the transaction, it is distributed to other participants with a flooding algorithm. Participants willing to provide computational power for the

PoW, the so-called miners, gather new transactions, put them into a block and try to find a hash value that is below a certain a threshold. The hash value is calculated from the block header, which contains a nonce that is adapted for each try. Once the miner has found a valid hash value, the new block is spread in the network. If more than one block has been mined, the participants choose the longest blockchain, i.e. the blockchain with the highest mining difficulty. The mining difficulty is inversely proportional to the threshold: the higher the mining difficulty, the lower the threshold. Since the network's overall computational power increases steadily, the mining difficulty is increased, too. To compensate their work, miners receive a reward in the form of Bitcoin units.

No central authority is involved in the payment process and the management of the network. Instead, the system is based on strong cryptography and the trust in a majority of the participants to follow the Bitcoin protocol. However, Bitcoin comes with several drawbacks. Since every transaction is publicly visible in the Bitcoin network, it is possible to trace the cash flow of a single participant. As a mitigation, participants can generate arbitrarily many new key pairs and therefore addresses to obfuscate their payment history. Another downside is the network's resource consumption: the mining process consumes a significant amount of time and energy. Consequently, it can take several minutes for a Bitcoin transaction to be processed. For a transaction with a substantial amount of Bitcoin units, it is even recommended to wait up to one hour.

Ethereum

Ethereum is a distributed smart contract platform proposed by Wood et al. in 2014 [64]. It is based on a public distributed ledger that is implemented with a blockchain. At the time of writing this thesis, the consensus on new blocks is reached via PoW, but a migration to PoS is planned for the near future.

Although Ethereum has got its own currency, called *Ether*, it is more than just a cryptocurrency platform: the network offers the ability to execute *smart contracts*. A smart contract is a computer program that verifies and enforces a kind of digital contract. Typical examples include the handling of a fund-raising campaign, where funds are collected from different investors and payed out to the recipient only if a certain threshold has been exceeded, or a digital voting system to reach a decision on a certain topic.

In the Ethereum network, an *account* can either be controlled by a user or by a contract. Users can transfer Ether among each other, but also to a contract and thereby trigger its execution. The amount of Ether needed to trigger the execution depends on the contract, in particular on its complexity and number of operations, and can be seen as

the execution fee that is credited to the miners. A contract is able to store arbitrary data, transfer Ether to other accounts and even interact with other contracts. Contracts are stored in the ledger and run by all participants of the network using the Ethereum Virtual Machine (EVM). Typically, they are written in *Solidity*, a Turing-complete programming language specifically designed for that purpose. Figure 2.4 shows a minimal example of a smart contract in Solidity.

```
1  pragma solidity ^0.4.0;
2
3  contract SimpleStorage {
4      uint storedData;
5
6      function set(uint x) public {
7          storedData = x;
8      }
9
10     function get() public constant returns (uint) {
11         return storedData;
12     }
13 }
```

Figure 2.4: Example of a smart contract written in Solidity, storing a single unsigned integer that can be read and written by everyone

Aside from Ethereum's ability to execute smart contracts, there are other advantages over systems like Bitcoin. The latency for creating a new block is considerably lower and in the range of a dozen seconds. Moreover, the fees for an Ethereum transaction depend on its computational complexity and storage requirement, as opposed to Bitcoin fees that are primarily determined by the block size. However, there are also drawbacks of Ethereum: smart contracts must be run by all participants of the network, leading to a multitude of redundant computations.

MultiChain

MultiChain is a private and permissioned distributed ledger system introduced by Greenspan in 2015 [27]. Different ledger instances can be created for different purposes and participants to achieve separation of concerns and to guarantee confidentiality. MultiChain is implemented with a blockchain and compatible with the Bitcoin ecosystem. In particular, a MultiChain node can act as a node in the Bitcoin network.

It does not a provide a cryptocurrency out of the box, but allows to issue and transfer custom *assets*. An asset can represent any quantifiable resource, like a digital currency, and can be enriched with arbitrary metadata. Additionally, MultiChain offers key-value based data storages called *streams*. Once a stream has been created, interested

participants can subscribe and write to it. In order to efficiently access the contents of a stream, an index is created on the node of each subscriber.

MultiChain comes with a fine-granular permission management. Among others, there are permissions for administrating, mining, connecting to a network, sending and receiving transactions, issuing assets, creating streams, and writing to a stream. Ordinary transactions with special metadata are used for granting and revoking permissions. It is even possible to grant permissions temporarily, i.e. for a specific number of blocks. A certain proportion of administrators has to agree before a permission change can take place. Different configuration parameters define the proportion required for each permission type.

The mining permission is the basis for the blockchain's consensus model. Although every participant with this permission is able to create a new block, none of them can monopolize the mining process. A parameter called *mining diversity* with a range between zero and one determines how the creation of new blocks is distributed among the miners. For that, a parameter called *spacing* is calculated by multiplying the mining diversity with the number of miners and rounding up. If the miner of the new block mined one of the previous *spacing*-1 blocks, the new block is invalid. Due to its permission management, MultiChain requires no PoW for the block consensus. Moreover, transaction fees and mining rewards are set to zero by default, but can be adjusted for each ledger instance individually, if desired.

MultiChain can be seen as a private and decentralized database. Since new blocks are created by a set of trusted actors, consensus algorithms like PoW become obsolete. Therefore, the frequency for creating new blocks can be significantly higher than in systems like Bitcoin or Ethereum. However, the use cases of private and permissioned ledgers differ and are restricted to environments with preselected participants.

Hyperledger

Hyperledger is an international umbrella project for DLT and was initiated by the Linux Foundation in 2015. Its subprojects are released under Free and Open-Source Software (FOSS) licenses and focused on corporate blockchain technologies. In the following, we present two of the most important projects.

Sawtooth: Hyperledger Sawtooth is an *"enterprise blockchain platform for building distributed ledger applications and networks"* [44]. One of the core features is its smart contract engine that supports a variety of languages, including Go, JavaScript and

Python, and that even can be used for making decisions on the blockchain's configuration. Furthermore, smart contracts written in Ethereum's Solidity are supported. To increase the block creation frequency, Sawtooth's transaction execution engine allows to process multiple transactions in parallel.

The consensus model can be exchanged dynamically at runtime. As a unique feature, Sawtooth supports Proof of Elapsed Time (PoET), a novel approach to achieve distributed consensus in an efficient way [30]. It utilizes a trusted execution environment to conduct a leader election. Each competitor samples a random value and waits for an amount of time determined by that value. The competitor with the smallest sample becomes the leader and is allowed to create the next block. In order to prevent competitors from forging the sample, the whole process is executed in a hardware-protected environment. For verifying that the competitor effectively waited the required amount of time, the protected environment provides a cryptographically secured proof.

As PoET was initially proposed by Intel, it is currently based on their Software Guard Extensions (SGX) supported by several Intel CPUs. However, new problems arise with this solution: all participants have to trust the manufacturer, in this case Intel, introducing a central authority for a technology aimed at decentralization.

Fabric: Hyperledger Fabric is a framework for operating a private and permissioned distributed ledger with a focus on modularity and extensibility [2, 51]. The building blocks of a Fabric network are peer nodes that execute smart-contract based transactions and an *ordering service* that broadcasts a block of ordered transactions. These transactions affect the ledger state stored on the peer nodes.

Consensus The consensus model of Fabric consists of three subsequent phases: the *endorsement* phase is driven by an *endorsement policy* upon which peers endorse a transaction, while the *ordering* phase determines the order of transactions to be committed to the ledger. Finally, the *validation* phase verifies the correctness of the endorsement policy and the transactions' results. Fabric allows to precisely adjust the configuration and behaviour of each of these phases.

Chaincode The business logic of an application running on top of a Fabric network is written in *chaincode*, which is similar to a smart contract [38]. Currently, it is only possible to implement chaincode in Go and JavaScript, but support for other languages like Java will be available in the near future. Figure 2.5 shows a simple example of a chaincode implementation in Go. A chaincode is installed on a set of *endorsing peers*,

together with a dedicated *endorsement policy* that specifies which peers have to *simulate* and *endorse* any transaction on the chaincode.

```go
package main

// SimpleAsset implements a simple chaincode to manage an asset
type SimpleAsset struct{}

func (t *SimpleAsset) Init(stub shim.ChaincodeStubInterface) peer.Response {
    // Initialize the asset
    // ...
    return shim.Success(nil)
}

func (t *SimpleAsset) Invoke(stub shim.ChaincodeStubInterface) peer.Response {
    fn, args := stub.GetFunctionAndParameters()

    var result string
    if fn == "set" {
        result, _ = set(stub, args)
    } else {
        result, _ = get(stub, args)
    }

    return shim.Success([]byte(result))
}

func set(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    stub.PutState(args[0], []byte(args[1]))
    return args[1], nil
}

func get(stub shim.ChaincodeStubInterface, args []string) (string, error) {
    value, _ := stub.GetState(args[0])
    if value == nil {
        return "", fmt.Errorf("Asset not found: %s", args[0])
    }
    return string(value), nil
}
```

Figure 2.5: Example of a Fabric chaincode in Go, implementing a simple asset accessible by a key, with operations to *set* and *get* it

Transaction Flow Figure 2.6 illustrates the flow of a transaction in a Fabric network, based on the official documentation [49]. A transaction creates or modifies a data record stored in the ledger, which consists of a blockchain and a state database. To modify the ledger, a client *invokes* a certain operation of the chaincode by sending a *transaction proposal* to the endorsing peers that have the chaincode installed (1). Each of them simulates and endorses the operation (2), i.e. without modifying the ledger, and returns the signed execution result in the form of an *endorsement* to the client (3). If all results match, the client will send the transaction together with the set of endorsements to the *ordering service* (4), a composite of several independent nodes that gathers transactions

from all clients and eventually puts them into a block that is delivered to all peers in the network (5). Each peer receiving the block, now in the role of a *committing peer*, appends the whole block to its blockchain copy. Afterwards, the peer checks the validity of each transaction in the block, i.e. if the endorsement policy is fulfilled, the execution results of the different peers match and no double-spending or read-write conflict has taken place. If this holds, the peer will apply the transaction's execution result to its copy of the state database (6).

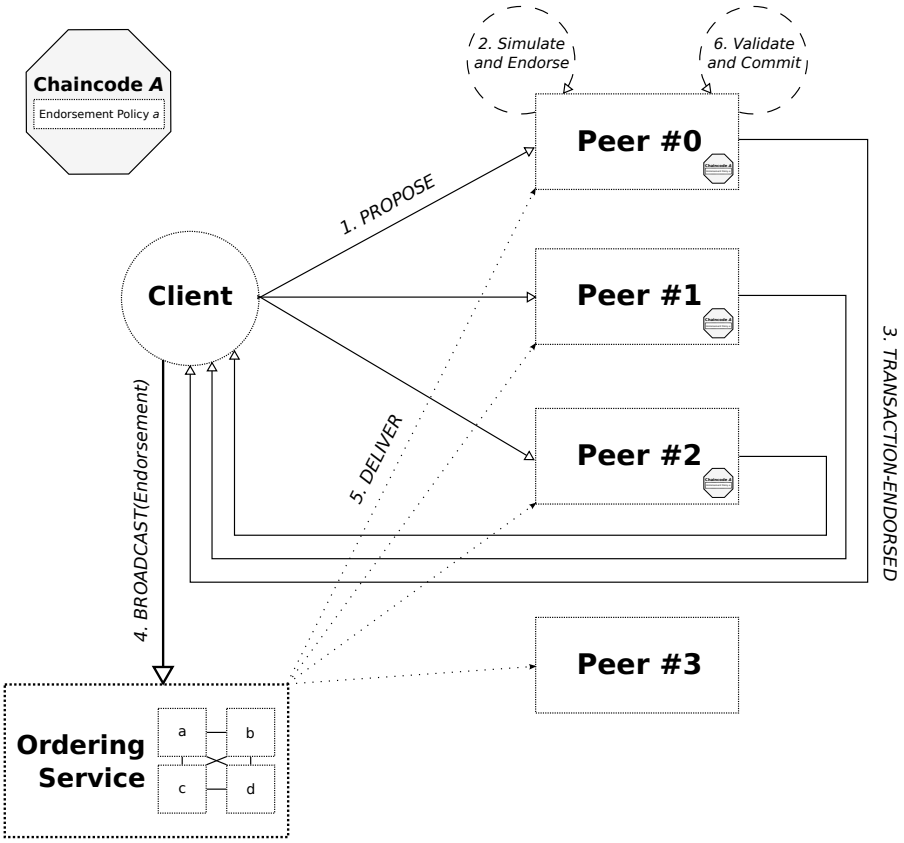


Figure 2.6: Transaction flow in a Fabric network, starting with a transaction proposal of the client (1), followed by the simulation and endorsement of the endorsing peers (2), their single endorsements returned to the client (3), the set of endorsements broadcasted to the ordering service consisting of several orderer nodes (4), the delivery of a new block including the transaction to the peers (5), and finally the transaction's validation and commitment (6)

Endorsement Policy An endorsement policy defines the requirements for a transaction on a chaincode to be successfully endorsed. It is is described by *principals* and *threshold gates* [42]. A principal corresponds to an entity that is expected to endorse a transaction, such as an endorsing peer. A threshold gate is defined by an integer

threshold and either a set of principals or another set of threshold gates. The threshold itself indicates how many principals or nested gates have to be satisfied. There are also shorthand notations for conjunctions and disjunctions. Figure 2.7 shows some typical examples of endorsement policies.

```
# A peer of Org1MSP and a peer of Org2MSP must endorse
T(2, 'Org1MSP.peer', 'Org2MSP.peer')
# Shorthand
AND('Org1MSP.peer', 'Org2MSP.peer')

# Either a peer of Org1MSP must endorse,
# or a peer of Org2MSP and a peer of Org3MSP
T(1, 'Org1MSP.peer', T(2, 'Org2MSP.peer', 'Org3MSP.peer'))
# Shorthand
OR('Org1MSP.peer', AND('Org2MSP.peer', 'Org3MSP.peer'))
```

Figure 2.7: Exemplary endorsement policies for a Fabric chaincode

Membership Service Provider (MSP) For managing identities and permissions, Fabric provides the concept of a Membership Service Provider (MSP) [45]. Each stakeholder participating in a Fabric network can be represented by a separate MSP.

An MSP defines governance rules for validating identities on the one hand and for generating and verifying signatures on the other hand. It represents a *member*, i.e. a legally separate entity such as a consortium, an organization or an Organizational Unit (OU). In order to validate identities based on a Public Key Infrastructure (PKI), the MSP is associated with at least one Certificate Authority (CA). Different configuration parameters can be specified for that purpose, like self-signed certificates for root CAs, certificates for intermediate CAs, and Certificate Revocation Lists (CRLs). Generally, an identity is assumed to be valid for an MSP if its certificate has a valid path to one of the root CAs and is not included in any CRL [46].

Channels A channel is a private subnet and can conceptually be seen as a completely separated ledger instance [40]. It consists of members, joined peers, connected clients, contributed orderer nodes, chaincode applications and the ledger itself. The participants must be authenticated and authorized to transact on a channel. Each channel provides confidentiality between the participants.

For each channel a member subscribes to, the member owns a set of peers. At least one of these peers serves as an *anchor peer* that the other peers communicate with to discover all participants of the channel. Furthermore, the member algorithmically elects a *leading peer* that communicates with the ordering service on behalf of the member.

Ordering Service The ordering service orders transactions into a block [36]. It consists of orderer nodes that may belong to different channels and typically operates network-wide, i.e. for different channels. Its implementation is easily exchangeable.

Two implementations are currently available. On the one hand, there is a *solo* ordering service, consisting of a single orderer node. It is recommended for testing purposes only, since it is a single point of failure. On the other hand, there is an implementation based on Apache Kafka, a distributed publish-subscribe stream processing platform with a focus on crash tolerance and high throughput.

At the time of writing this thesis, an ordering service implementation providing Byzantine Fault Tolerance (BFT) is under development, but not yet finished. Currently, only a proof-of-concept implementation of a BFT ordering service is available [62]. It employs several orderer nodes and can cope with less than one third of faulty nodes. The prototype was contributed by an independent team of developers from the University of Lisbon and is based on their free Java library *BFT-SMaRt* [11], which focuses on the achievement of high performance similar to the Practical Byzantine Fault Tolerance (PBFT) approach by Castro et al. [14].

Chapter 3

Analysis

This chapter analyzes and describes the basic problem that we aim to address with our work. To solve the problem, we present the functional and non-functional requirements for a novel configuration management process and propose different approaches for designing and implementing it.

3.1 Basic Problem

As the management and administration of an IT environment is critical with respect to safety and security, it is crucial to base this discipline on a sophisticated and technically mature authentication and authorization concept. Consequently, only administrators with a professional knowledge and experience should be allowed to make significant changes to a system. However, the assignment of administrative privileges carries serious risks: without further precautions, administrators typically have comprehensive control over the system they are responsible for, with the danger of causing severe damage to it. On the one hand, administrators can make accidental faults. On the other hand, damage can be caused deliberately, either by a disloyal administrator or by an external attacker that successfully has compromised an administrator's account.

In the recent past, more and more attention has been drawn to the fight against *insider threats* [61]. Although there are varying definitions for this kind of threat, a common characteristic is the misuse of privileges granted to a current or former employee or associate [29, sec. 1.2]. The attacker's motivation can be diverse, ranging from the frustration after having been suspended to the espionage on behalf of a competitor. Since insiders have a detailed knowledge of the organization's IT environment and usually do

not need to bypass a firewall or an Intrusion Detection System (IDS), these attacks are hard to detect. A multitude of countermeasures is required to effectively prevent and defeat them, limited not only to technical solutions like access control and monitoring, but also covering sociological and organizational approaches like reducing the motivation for attacks and establishing a different corporate culture [29, sec. 5].

In contrast, external attackers try to infiltrate an organization by compromising employees who are assigned administrative privileges. For that, they specifically target an individual victim with sophisticated methods, typically by implanting tailored malware. After the attackers have gained access to the victim's account, they abuse the administrative privileges for a lengthy period of time in order to steal critical data or to cause damage to the organization. This attack model, also known as Advanced Persistent Threat (APT), was first described by the United States Air Force in 2006 [12, p. 3]. Many of these attacks are based on social engineering. While *spear phishing* deals with highly personalized phishing e-mails, *water holing* is based on the manipulation of websites the victim regularly visits. Another approach called *baiting* involves actions in the real world: attackers leave physical media like memory sticks infected with malware at locations where the victim is likely to find it, expecting that it will be plugged in a worthwhile target computer.

The fundamental problem concerning both kinds of attacks is the involvement of administrative privileges. These privileges do not only allow to read critical data and to modify configurations, but also often provide the ability to circumvent or disable the security measures established to secure the IT environment, like access control, logging mechanisms, an Intrusion Detection System (IDS) or solutions for Security Information and Event Management (SIEM). If the administration of the IT environment is not distributed to several administrators, the access to a single administrator account will presumably suffice to perform a successful attack with severe consequences.

3.2 Requirements

To mitigate the problem described in section 3.1, we present a configuration management process that restrains the power of a single administrator and distributes it to a quorum of administrators in a tamper-resistant way. In the following, we formulate the functional and non-functional requirements for this process.

3.2.1 Multi-Party Authorization

For conducting the configuration management process, we leverage the principle of Multi-Party Authorization (MPA), as described in section 2.2. Thereby, each configuration must undergo a validation procedure, where it is reviewed and approved by a set of independent experts. Only after a configuration has been successfully validated, it is applied to its targets.

The conditions required for a configuration to become valid can be specified per target, making it possible to take into account the different criticalities of targets. Depending on the specified conditions, multiple parties have to reach a consensus on the validity of a configuration, for instance by requiring m out of n experts to review and approve a configuration, with $n > m > 1$. That way, faulty or malicious configurations can be effectively revealed and averted.

3.2.2 Authentication

As a prerequisite for enforcing authorization, all subjects that are involved in the configuration management process have to be authenticated. This covers the confirmation of the subject's identity as well as the actual authentication mechanism, preferably taking into account multiple authentication factors like knowledge and possession.

3.2.3 Accountability and Non-Repudiation

All subjects that are involved in the configuration management process must be accountable for their actions. This includes the principle of non-repudiation, preventing subjects from denying their actions in hindsight.

On a technical level, accountability and non-repudiation are ensured using asymmetric cryptography: each action must be digitally signed by the corresponding subject.

3.2.4 Traceability

To understand which subject configured which target in what way, all actions of the configuration management process must be traceable via an event history. For every action, it is required to record its initiator and its date and time. This applies to the submission as well as to the approval of configurations.

3.2.5 Tamper-Resistance

The configuration management process must be conducted in a tamper-resistant way. No attacker must be able to manipulate the input, the execution or the output of the process. Besides, manipulation attempts must be able to be detected reliably.

Protected Execution

The execution of the configuration management process must take place in a protected environment that is resistant to manipulations and intrusions. Even the abuse of administrative privileges must not be able to influence the execution of the process.

Unerasable and Unforgeable Storage

All data related to the configuration management process must be stored in an unerasable and unforgeable way. For this purpose, the creation, modification and erasure of data records is exclusively controlled by the configuration management process that is executed in a protected environment. No data records must be created, modified or erased unauthorizedly. In particular, it must not be possible to erase data records from the event history, but only append records to it.

3.2.6 Locking

To prevent administrators from bypassing the configuration management process, all managed targets must be locked appropriately. In particular, no one must be able to directly access the targets with administrative privileges. Adequate technical measures include, among others, the shutdown of network services that allow remote logins. However, a direct access in case of emergency may still be provided.

3.2.7 Resilience

The configuration management process should be resilient to attacks and failures. In particular, a temporary outage of a subset of components should not disrupt the overall service. Moreover, errors occurring during the configuration management process should be handled gracefully and not result in an undesired and inconsistent state.

3.2.8 Automation

To a certain extent, the configuration management process should be automatable. Therefore, an Application Programming Interface (API) should be provided that is conveniently accessible for other software components, such as a Continuous Configuration Automation (CCA) service. The degree of automation depends on the specific use case. Typically, it is reasonable to apply valid configurations to the targets automatically. However, further steps of the process may be automated, either entirely or partially, like the approval of a configuration.

3.2.9 Interoperability

It should be easy to integrate and embed the configuration management process into the existing IT landscape of an organization. On the one hand, this refers to the management of identities. Identities and roles needed for conducting the configuration management process, like administrators and target devices, may be provided by an existing identity management system. On the other hand, this covers the interaction with existing configuration management frameworks, as introduced in section 2.1. It is reasonable to utilize their powerful script languages to express configurations in a human- and machine-readable way. Moreover, these frameworks can act as a post-processor for our configuration management process: after a configuration has been validated in a tamper-resistant way, it is handed over to such a framework in order to actually apply it to a target.

3.3 Design Choices

Different approaches can be chosen for designing and implementing the configuration management process. The centerpiece of this work is to leverage Distributed Ledger Technology (DLT) for that purpose. Subsequently, we specify the possible choices in greater detail.

3.3.1 Process Design

The configuration management process can be designed in several ways. Generally speaking, it consists of a sequence of process steps. Each process step is initiated by a subject, for instance an administrator, and has a clearly defined set of inputs and outputs, leading to a state change of an object, for instance a configuration. To design

and document the process, purpose-built description languages can be used, like Event-Driven Process Chains (EPCs) and UML Activity Modeling.

In order to achieve MPA for configurations, two process steps are of particular importance: *propose* a new configuration one the one hand and *approve* it on the other hand. That way, multiple parties are able to authorize a configuration. Depending on the specific use case, further process steps may be useful, for instance to explicitly disapprove a configuration or to handle its activation on a target.

3.3.2 Distributed Ledger Technology

Distributed Ledger Technology (DLT) can provide the basis for conducting the configuration management process trustworthily. In contrast to a centralized solution where the users have to trust a single operator, DLT allows to perform the process in a decentralized and distributed way, preventing an individual participant from being able to manipulate or forge the process and its outcomes.

This can be achieved in two respects: on the one hand, the distributed ledger stores the inputs and outputs of the process. An invocation of the configuration management process leads to a new transaction, which will be included in the ledger as soon as a new consensus has been reached. Due to the characteristics of DLT, this takes place in an unerasable and unforgeable manner: DLT implementations like blockchains guarantee that existing transactions are neither erased nor modified, while the redundant and distributed peer-to-peer architecture prevents manipulations and single points of failure.

On the other hand, DLT can provide means to execute the process in the peer-to-peer network itself, typically in the form of a smart contract. With that approach, the distributed ledger does not only manage the inputs and outputs of the process, but also its procedural implementation and execution. Consequently, the transparency, trustworthiness and legal compliance of the process are increased.

With DLT, it is possible to meet the security requirements defined in section 3.2, including accountability, traceability and tamper-resistance. As a beneficial side effect, the peer-to-peer network of a distributed ledger can also contribute to assure resilience.

3.3.3 Identity Management

Before subjects of an organization can participate in our configuration management process, they need to be authenticated. To reach this goal, there are two basic ap-

proaches: either by establishing a separate identity management system particularly for the configuration management process, or by reusing and connecting an existing one.

Most organizations operate a centralized identity management system that confirms identities based on personal data, assigns roles and permissions, and implements the actual authentication mechanisms, for instance by using public key cryptography. In practice, the design and implementation of such a system can vary considerably: while some organizations run a simple directory service accessible via LDAP, others manage a whole Public Key Infrastructure (PKI) with internal Certificate Authorities (CAs).

To integrate such an identity management system into our process, both components have to communicate over a secured and clearly defined interface. Moreover, it may be advisable to create specific roles for the subjects participating in the configuration management process, like roles for approvers and targets.

However, the implications of using an existing identity management system have to be considered carefully. If an individual administrator has write access to the identity management system, it will be trivial to circumvent the Multi-Party Authorization (MPA) established by our process, simply by creating new identities or modifying existing ones as needed.

3.3.4 Configurations

In practice, administrators have to deal with a wide variety of configuration types and formats. In our work, we want to manage configurations of arbitrary types and formats, independent of specifics of the managed systems and environments. In the following, we describe different approaches for that purpose.

Format and Type

Commonly, configurations represent a system's state and behaviour in a human- and machine-readable format. However, many systems and applications define own formats and semantics for their configurations. Typical examples include the custom configuration format of the *Apache HTTP Server* that is based on a large set of configuration directives, and the table format of the Linux packet filter *iptables*. Although there are efforts for establishing standardized serialization formats for configurations, like XML and YAML, the interpretation of the semantics of a configuration often remains application-specific.

By introducing an explicit typing of configurations, it is possible to manage configurations of different formats and for different applications. For that, each configuration is enriched with a type and further metadata necessary to handle the configuration and to differentiate the configuration management process appropriately. With reference to the examples above, it is then possible to declare separate configuration types for Apache HTTP Server and iptables. In addition, it may be useful to introduce dedicated types for meta configuration formats, like Chef recipes and Ansible playbooks.

Activation

After a configuration has been authorized using our configuration management process, it is finally applied to its targets. In a modest way, this can be achieved without the assistance of comprehensive toolsets, for instance by simply writing a configuration file to its intended location or by executing a configuration shell script. However, these approaches quickly reach their limits, as they can hardly provide error handling and recovery, trigger events for pre- and post-processing, and guarantee to always leave the target in a consistent state.

As soon as the complexity of the configurations and the managed systems increases, it is recommended to employ configuration management frameworks, as introduced in section 2.1. They do not only solve the problems mentioned before, but also offer plenty of additional features, like declarative configuration approaches and the management of whole IT landscapes. Furthermore, they provide unified languages for managing configurations on a meta-level, supporting many applications and environments with different configuration formats at one go.

For that purpose, our configuration management process must be able to communicate with such a framework via a well-defined interface. In particular, the configuration management framework should be triggered automatically once a configuration has been authorized by our process.

Chapter 4

Related Work

An integral part of any management framework is a proper concept for authentication and authorization. That way, only subjects with the appropriate permissions can make changes to the managed system, resulting in a higher level of security and accountability. To limit the power of individual subjects, organizations can employ the principle of redundant supervision, for instance by implementing Multi-Party Authorization (MPA) as described in section 2.2. In the following, we take a look at existing solutions for reviewing and approving changes in such a way, with an overview given in table 4.1.

Although a majority of configuration management frameworks provides mechanisms for access control, most of them do not offer an integrated review and approval process for configurations [20, sec. 3.3.6 a. 3.3.7]. For some frameworks, commercial extensions are available that allow the enforcement of generic workflows, like Ansible Tower [5]. However, these workflow engines are not specifically designed for reaching a Byzantine fault-tolerant consensus on configuration changes.

Vanbrabant et al. present a workflow enforcement system called *ACHEL* [63] that utilizes a Distributed Version Control System (DVCS) such as Git. ACHEL is able to establish an authorization workflow for configurations administered with a configuration management framework like Chef, Puppet or Ansible. For modelling access control rules and authorization workflows, ACHEL provides a custom-built but yet powerful policy language, which is even able to consider the semantics of configuration changes. Each participant, that is either represented by a user or a configuration management framework itself, controls a dedicated repository. If a user wants to conduct a configuration change that requires an authorization workflow, she adds that change to her repository and contacts other users via an existing communication channel, like e-mail

or instant messaging. These users are defined in a policy and serve as authorizers. Thus, they are responsible for reviewing the change. If they agree, they will sign the change and add it to their individual repositories. The requesting user will then collect these signatures and push it to the repository of the configuration management framework, which will verify if the conducted authorization workflow conforms to the respective policy.

There are two major conceptual differences between ACHEL and our work: on the one hand, we employ a distributed and tamper-resistant ledger with a global state and a comprehensive log of all configuration-related data records and operations. In contrast, ACHEL's design is based on separate repositories per user that are not synchronized. Furthermore, each authorized configuration ends up in the repository of the configuration management framework, which acts as a single and centralized database and in turn can be manipulated by its administrator. On the other hand, ACHEL is based on a different trust model: in contrast to our work, ACHEL requires a single party to govern and operate the whole system, which all participants need to trust in. Therefore, it provides no means to manage configurations for infrastructure shared across participants that only share a limited amount of trust, like different organizational units or a consortium of several organizations.

In the field of Change Management, the term *change control* refers to a process that governs the introduction and implementation of changes for a managed environment. It is closely connected with the fields of Quality Management (QM) and Project Management (PM). Consequently, the change control process is not restricted to software and configuration changes, but also includes organizational and regulatory changes. According to a definition of the Council of Europe, change control *"ensures that changes are appropriately identified, planned, documented, validated (where relevant), approved, verified and traceable"* [22, sec. 4], with the goal to *"prevent risks to the quality and safety of results"* [22, sec. 1]. However, most change management frameworks are released under non-free licenses and intended for commercial purposes only. Furthermore, they are not tailored to the management of software configuration changes, but cover the whole spectrum of IT Service Management (ITSM). A prominent example, with a feature-limited edition released under the GNU Affero General Public License (AGPL), is the OTRS framework [56].

A completely different approach for conducting an auditing process is described by a formal concept called *transparency overlays*. It was proposed by Chase and Meiklejohn in 2016 [15]. A transparency overlay can be operated on top of any IT system in order to make its actions publicly auditable. Concrete examples include Certificate Transparency (CT) [34], a framework for auditing certificates issued by a Certificate Authority (CA),

and the Bitcoin payment system. CT became increasingly popular as a defense against rogue and compromised CAs after numerous media reports on the unauthorized issuance of certificates in 2011 [13].

For their approach, Chase and Meiklejohn define a Dynamic List Commitment (DLC), which represents a commitment to a list of arbitrary elements, typically in the form of a Merkle tree. The only way to modify the list is to append a new element to it. Furthermore, it can be proven that a given element is in the list. To actually run a transparency overlay on top of an existing *system*, one needs to operate a *log server*, an *auditor* and a *monitor*. In order to avoid a single point of failure, each component can be operated redundantly. The log server stores events produced by the system in a public log that is implemented with a Merkle tree. The auditor verifies that specific events are contained in the log, without needing to hold the entire log available. The monitor is responsible for identifying and reporting events of interest, for instance those that appear unusual or suspicious. Finally, auditor and monitor corporate to make subjects publicly accountable for their actions.

Based on this concept, Hof and Carle proposed a transparency and auditing system for the distribution of software updates, in particular for Debian's package management system *APT* [28]. Although the integrity of updates performed with APT is secured with cryptographic signatures, there are still attack scenarios on the update process that are hard to detect. One of them is the *targeted backdoor* attack that aims to deliver a correctly signed but manipulated update to a preselected victim, while all other users are served with the regular update. In that scenario, the package maintainers have little interest to reveal that they have been compromised. Since the attack leaves a trail in the log, however, it can be easily detected by auditors and monitors. As an additional feature, the transparency system is able to verify the binding between an update's source and binary code and thereby discloses faults and manipulations in the build process.

Although the security goals of transparency overlays and our work are similar, namely to ensure the compliance and accountability of a system's actions in a tamper-resistant manner, the approaches are quite different: while transparency overlays are focused on auditing actions in hindsight, our work deals with reviewing and approving actions before they can take place. This aspect, together with the foundation on Distributed Ledger Technology (DLT), make our work unique.

	Ansible	**ACHEL**	**Transparency Overlays**
Authorization	✓	✓	
Multi-Party Authorization	(✓)[1]	✓	
Auditability	(✓)[2]	✓	✓
Tamper-Resistance		(✓)[3]	✓
Distributed process execution and data storage		✓	
Logically centralized configuration repository			
Federated configuration management across stakeholders' boundaries			
Semantic interpretation of configurations		✓	
Resistance to compromisation of targets			

Table 4.1: Comparison of related work regarding the objectives pursued by our approach

[1] enforcement of generic workflows via plug-ins

[2] optional via built-in logging mechanisms

[3] attackers may be able to embezzle configuration proposals, since there is no logically centralized configuration repository

Chapter 5

Design

In this chapter, we introduce the high-level design of TANCS. It describes our system on a conceptual level and does not depend on a specific Distributed Ledger Technology (DLT), making it re-usable for different environments and implementations.

The chapter is structured as follows: first, we give an overview of the overall design, followed by a detailed description of our configuration management process. Then, we introduce the subjects and objects involved in our system. Lastly, we present the multi-tier architecture our system is based on.

5.1 Overview

Our system is designed to manage configurations based on the principle of Multi-Party Authorization (MPA). Consequently, a configuration is never applied to a target directly, but has to undergo a configuration management process controlled by our system first. The UML use case diagram in figure 5.1 gives an overview of the system interactions taking place on a conceptual level.

For conducting the configuration management process, our system stores *configuration requests* in an unforgeable, traceable and unerasable manner. A *configuration request* (CR) is a request to apply a certain configuration to a set of targets. The configuration itself is represented as a human- and machine-readable description of the targets' state.

Different subjects interact with our system to conduct configuration management operations. We provide operations to *propose*, *approve*, *retrieve* and *acknowledge* a CR.

These operations affect the state of the respective CR. As soon as the CR becomes *valid*, its configuration can be applied to its set of targets.

To verify the identity of the subjects, our system communicates with an external *identity provider*. An identity provider is responsible for authenticating subjects. For that, it may interoperate with existing IT systems, such as a Public Key Infrastructure (PKI). After the identity provider has authenticated a subject, our system checks if the subject is entitled to conduct the claimed configuration management operation.

For that purpose, authorization rules are defined by two different kinds of attribute-based policies: an *access control policy* (ACP) specifies who is allowed to propose a configuration for a set of targets, while a *validity policy* (VP) stipulates the requirements for a CR to become valid. A policy evaluator, which can be operated internally or provided by an external component, decides if an artifact fulfills a policy.

Our design involves another external component: in order to apply a valid configuration to a target, we utilize a *handler*. Typically, this handler is implemented by a configuration management framework, as introduced in section 2.1, and executed on the target itself. With that approach, even configurations based upon the features of such a framework can be managed with our system.

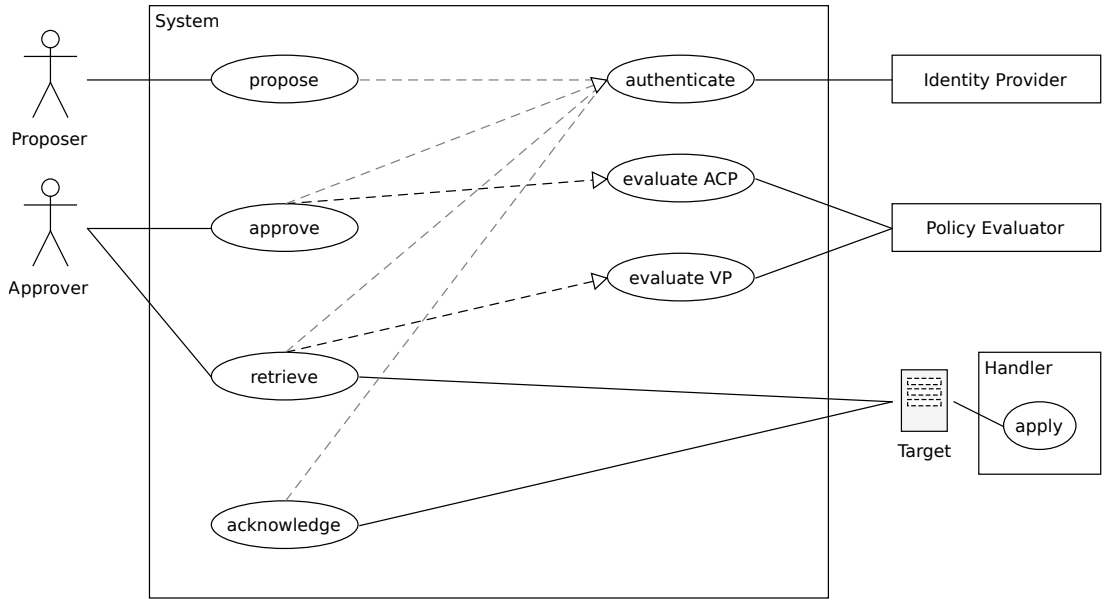


Figure 5.1: System interactions on a conceptual level, illustrated with an UML use case diagram

5.2 Configuration Management Process

The configuration management process comprises operations to *propose*, *approve*, *retrieve* and *acknowledge* a CR. In the following, each operation is described in detail and illustrated with an Event-Driven Process Chain (EPC).

5.2.1 Propose

A proposer *proposes* a new configuration together with a set of targets to the system, as depicted in figure 5.2. After the identity provider has successfully performed the authentication, the system checks if the subject is permitted to propose a configuration according to the *access control policy* (ACP) of the specified targets. If this holds, it will add a new CR containing the configuration to the repository. Initially, the state of the CR is set to *proposed*.

5.2.2 Approve

After a configuration has been proposed, an approver reviews the CR. If the review succeeds, the approver will *approve* the CR, as depicted in figure 5.3. Provided that the authentication of the subject has been successful, the system adds the approval to the CR. Then, it checks if the CR became valid according to the *validity policy* (VP) of the targets. If this holds, it will update the CR's state to *valid*.

5.2.3 Retrieve

Subjects *retrieve* CRs from the system, as depicted in figure 5.4. Internally, the matching CRs are read from the system's repository and returned to the inquirer. Different filters can be specified for retrieving CRs, for instance to match a certain state or a particular set of targets. The retrieval can be initiated manually or triggered by an event.

A typical use case includes targets that retrieve a valid CR destined for themselves in order to apply the configuration locally. Furthermore, approvers have to retrieve a CR before being able to review and approve it.

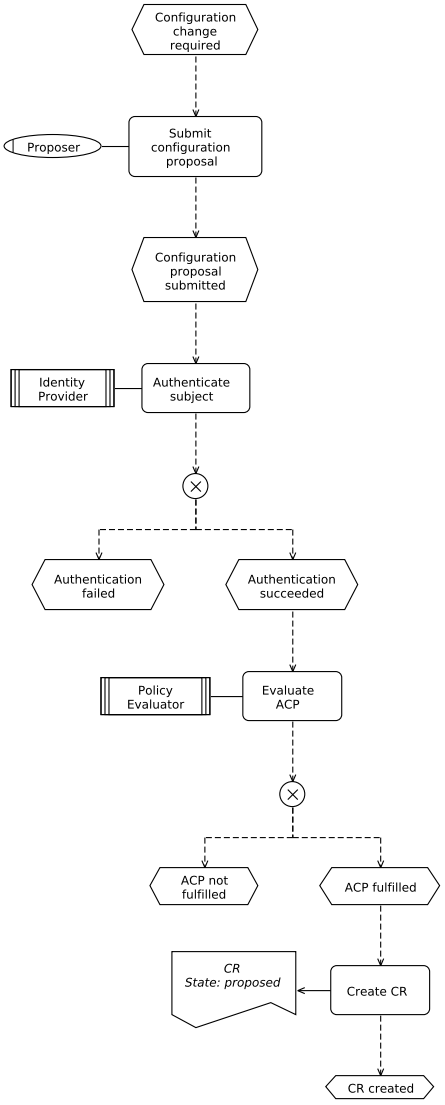


Figure 5.2: EPC model for *propose*

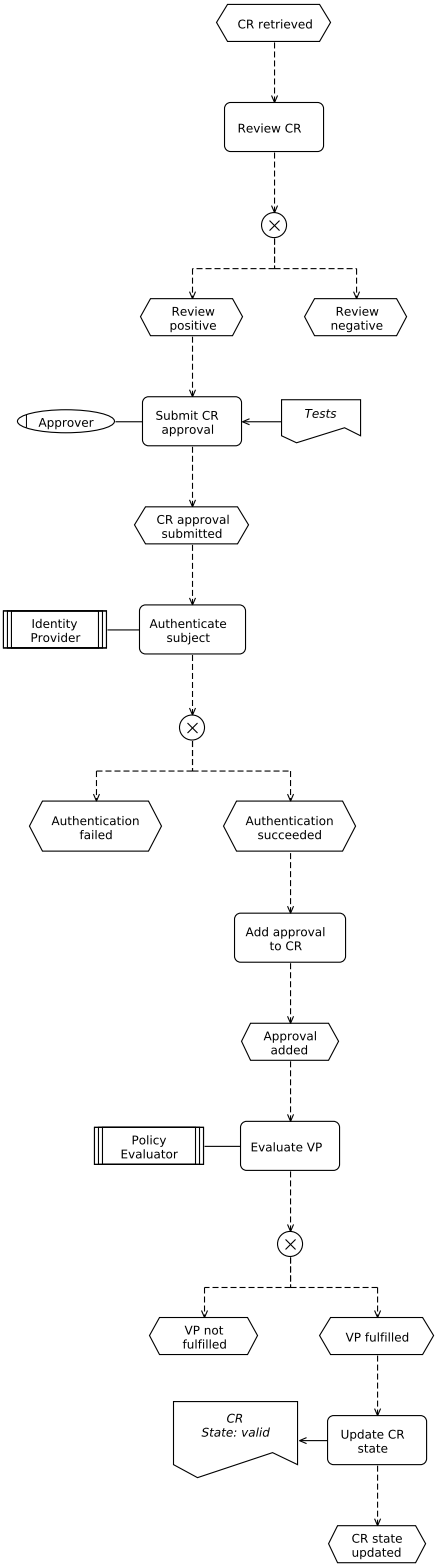


Figure 5.3: EPC model for *approve*

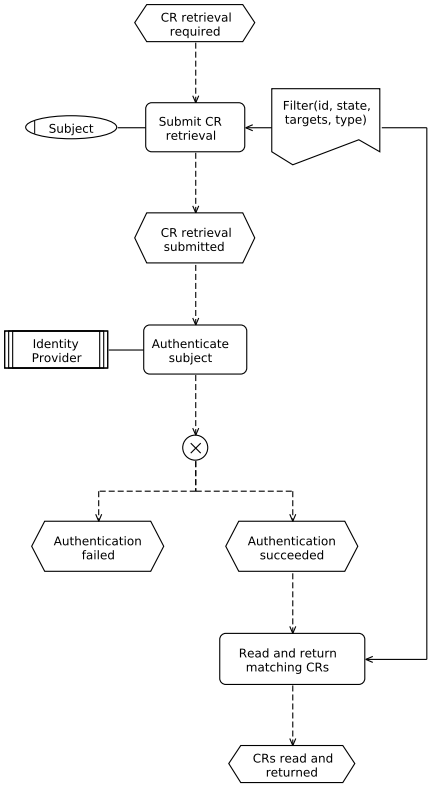


Figure 5.4: EPC model for *retrieve*

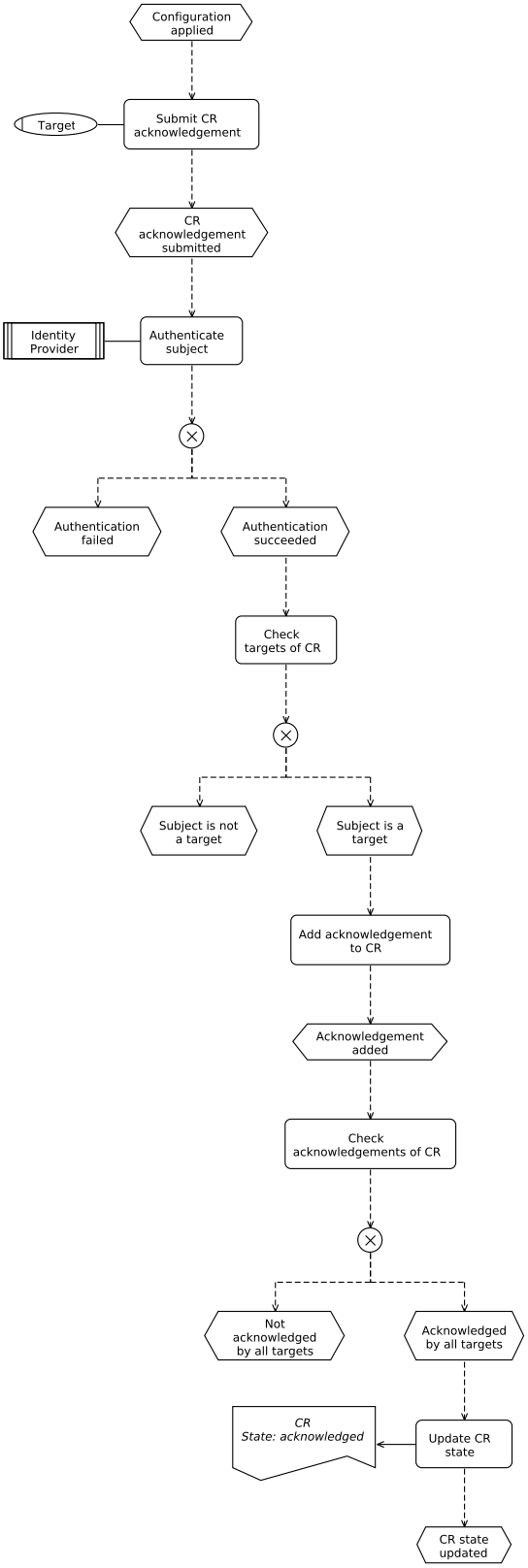


Figure 5.5: EPC model for *acknowledge*

5.2.4 Acknowledge

After having applied the configuration locally, each target *acknowledges* the CR, as depicted in figure 5.5. Internally, the system adds each acknowledgement to the CR. As soon as all targets have acknowledged it, the system updates the CR's state to *acknowledged*.

5.3 Subjects

Different kinds of subjects interact with our system. Understanding the relationships between these subjects is crucial for understanding the system's design and functionality.

5.3.1 Proposer

A *proposer* proposes a new configuration for a set of targets. Typically, an organization's administrator takes the role of a proposer. To submit the configuration to the system, the proposer has to be authenticated. Furthermore, the proposer needs to be explicitly authorized to address the specified targets.

5.3.2 Approver

An *approver* approves a configuration that has been proposed by a proposer beforehand. It is the responsibility of the approver to carefully review the configuration and perform tests on it, for instance that the configuration is syntactically and semantically correct or that it does not violate security guidelines.

Approvers may either be human experts or software components, the latter providing the possibility of semi or fully automated approvals. In practice, several approvals from different approvers are required to make a configuration valid.

5.3.3 Target

A *target* is the entity that has to be configured, for instance a workstation or server, also referred to as *managed device*. It does not matter if it exists physically or only virtually. It retrieves a valid configuration targeted to itself from the system and applies it locally. Afterwards, the target submits an acknowledgement to the system to state that it has applied the configuration.

5.4 Objects

Subsequently, we describe the different objects needed for the functionality of our system. Instances of these objects are stored in the system's repository.

5.4.1 Configuration Request

A *configuration request* (CR) is the essential object of our system. It is a request to apply a configuration to a set of targets. In the following, the structure and the lifecycle of a CR are described in detail.

Structure

Each CR has a unique identifier and holds its current state. Furthermore, it contains a proposal, a set of approvals and a set of acknowledgments. Figure 5.6 illustrates the internal structure of a CR with an Entity-Relationship Model (ERM), while figure 5.7 shows an exemplary instance of a CR represented in JSON.

Proposal: The proposal contains the actual configuration and the set of targets. It is time-stamped and linked to exactly one proposer. The configuration is represented by a serialization type and the serialized data itself.

Approval: An approval expresses that an approver has approved the CR. It is time-stamped and linked to exactly one approver. It may come with custom test attributes for stating that the approver has performed additional tests with respect to the CR. These attributes can be matched in a policy.

Acknowledgment: An acknowledgement expresses that a target has applied the configuration stored in the CR. It is time-stamped and linked to exactly one target.

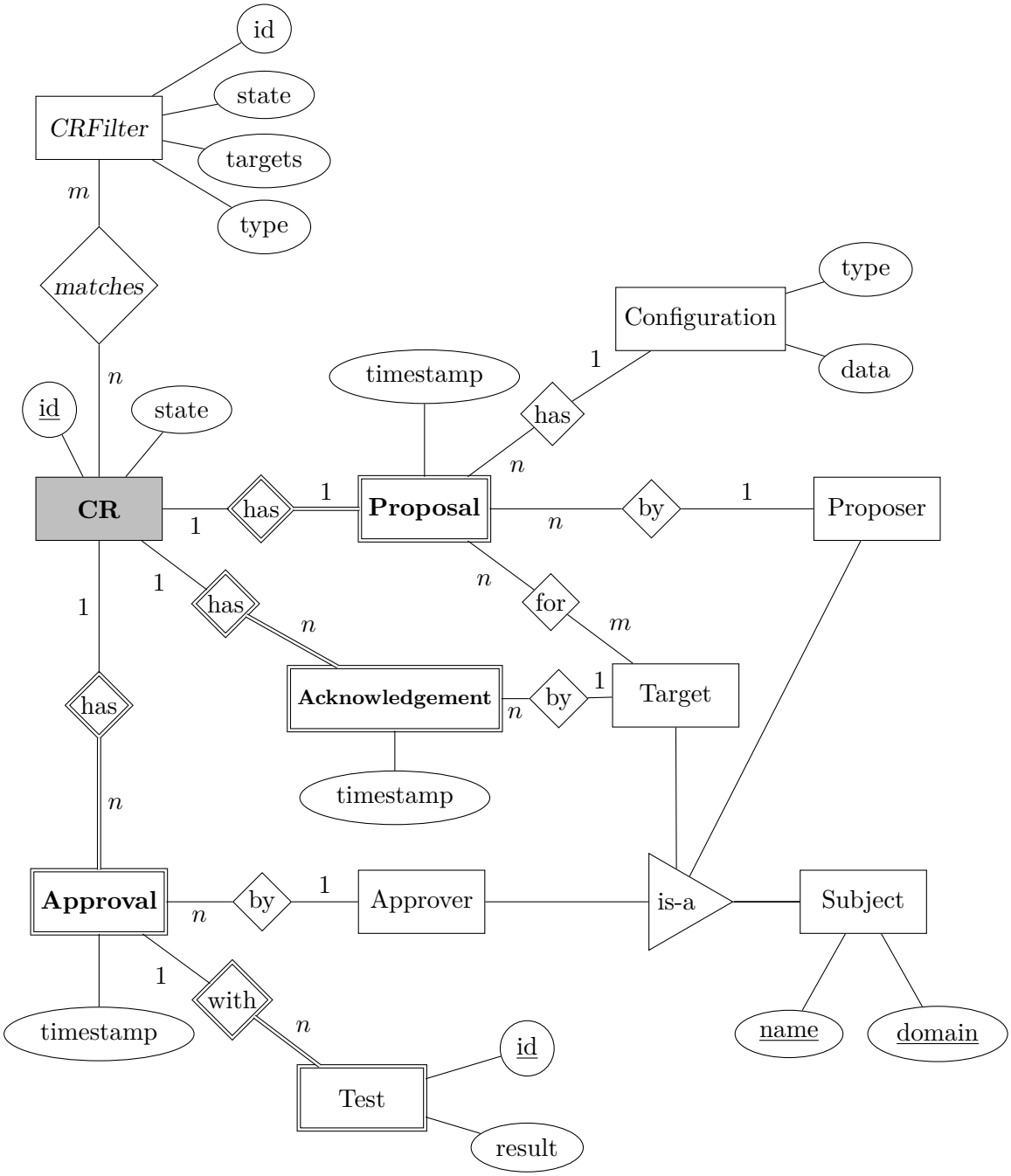


Figure 5.6: Structure of a CR, illustrated with an ERM

```
{
  "id": "CR_x",
  "state": "acknowledged",
  "latestStateChangeTimestamp": "2018-01-07T09:11:25Z",
  "proposal": {
    "timestamp": "2018-01-07T09:05:10Z",
    "proposer": {
      "name": "ProposerA",
      "domain": "Org1"
    },
    "targets": [
      {
        "name": "DeviceI",
        "domain": "Org1"
      }
    ],
    "configuration": {
      "type": "ansiblePlaybook",
      "data": "..."
    }
  },
  "approvals": [
    {
      "timestamp": "2018-01-07T09:07:15Z",
      "approver": {
        "name": "ApproverA",
        "domain": "Org1"
      },
      "tests": [
        {
          "id": "integrationTest",
          "result": "passed"
        }
      ]
    },
    {
      "timestamp": "2018-01-07T09:09:20Z",
      "approver": {
        "name": "ApproverB",
        "domain": "Org2"
      }
    }
  ],
  "acknowledgements": [
    {
      "timestamp": "2018-01-07T09:11:25Z",
      "target": {
        "name": "DeviceI",
        "domain": "Org1"
      }
    }
  ]
}
```

Figure 5.7: Example of a CR, represented in JSON

Lifecycle

Each CR has a lifecycle, expressed by the states *proposed*, *valid*, *acknowledged* and *outdated*. The UML state machine in figure 5.8 visualizes the state transitions of a CR induced by operations of our configuration management process.

The initial state of a CR is *proposed*, indicating that it has been submitted by a proposer, but not been processed any further. For each approver that approves the CR, a new approval gets added to the CR. As soon as the set of approvals fulfills the *validity policy* (VP) of the targets, the CR's state is updated to *valid*. Then, the targets apply the CR locally and acknowledge it, each time resulting in a new acknowledgement added to the CR. After all targets have acknowledged the CR, its status is updated to *acknowledged*.

As part of its initialization, each target is allowed to propose its initial ACP and its initial VP to the system. The corresponding CRs, and therefore both initial policies, become valid immediately, since there are no other policies to evaluate against.

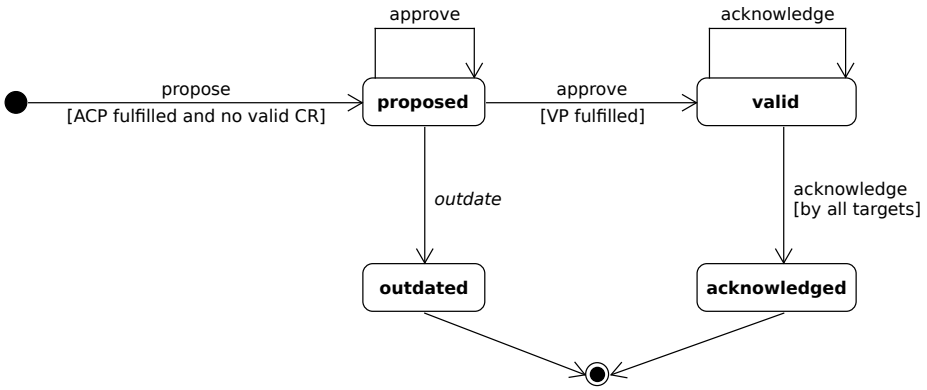


Figure 5.8: State transitions of a CR, illustrated with an UML state machine

To prevent race conditions and to ensure that each configuration is proposed with respect to a well-defined state of the targets, only one CR can become *valid* for the same set of targets at the same time.

This goal is achieved in two ways: on the one hand, if there are several CRs in the *proposed* state for a set of targets and one of them becomes *valid*, the state of the other proposed CRs for these targets will be set to *outdated*. Outdated CRs cannot be processed any further, but can still be retrieved from the system's repository.

On the other hand, further configurations cannot be proposed for the respective targets as long as there is a *valid* CR targeted to them. Only after this CR has been acknowledged by all of its targets and its state has transitioned from *valid* to *acknowledged*, new CRs can be proposed.

5.4.2 Policies

We introduce two different kinds of policies to describe the relationships between CRs, proposers, approvers and targets. Policies can be defined for a single target or a set of targets. Changes to policies are handled as ordinary CRs, guaranteeing the same principles that hold for normal configuration changes, like redundant supervision, accountability and traceability.

The definition and evaluation of the policies are based on Attribute-Based Access Control (ABAC). This concept allows to match ordinary attributes of entities with those defined in the policy, for instance the name and domain of a subject, providing a high degree of flexibility for almost any operational environment.

Access Control Policy

An *access control policy* (ACP) defines which subjects are allowed to propose a CR for which targets. Consequently, our system checks if a proposer is allowed to propose a CR according to the ACP of the targets. If the result is positive, the CR will be created.

Figure 5.9 illustrates the structure of an ACP in an Entity-Relationship Model (ERM), while figure 5.10 shows an exemplary instance. An ACP is specified for a set of targets and comes with a set of rules, each of them for a different configuration type. A rule contains a set of filters, each of them matching one or several proposers. The matching criteria are names or domains of proposers, or the conjunction of both.

Validity Policy

A *validity policy* (VP) defines which requirements have to be met for a CR to become valid. Each time a CR is approved, our system checks if the set of approvals satisfies the VP of the targets. If the result is positive, it will update the CR's status to *valid*.

Figure 5.11 illustrates the structure of a VP in an Entity-Relationship Model (ERM), while figure 5.12 shows an exemplary instance. Similar to an ACP, a VP is specified for a set of targets and comes with a set of rules, again each of them for a different configuration type. Each rule is linked to exactly one requirement.

Currently, there is support for a requirement that stipulates the fulfillment of m out of n filters, allowing to model a simple majority consensus with respect to the validity of a CR. Such a filter matches one or several approvals of a CR, based on the approver and the tests. For approvers, the matching criteria are names or domains, or the conjunction of both. A test is matched with respect to its identifier as well as its result.

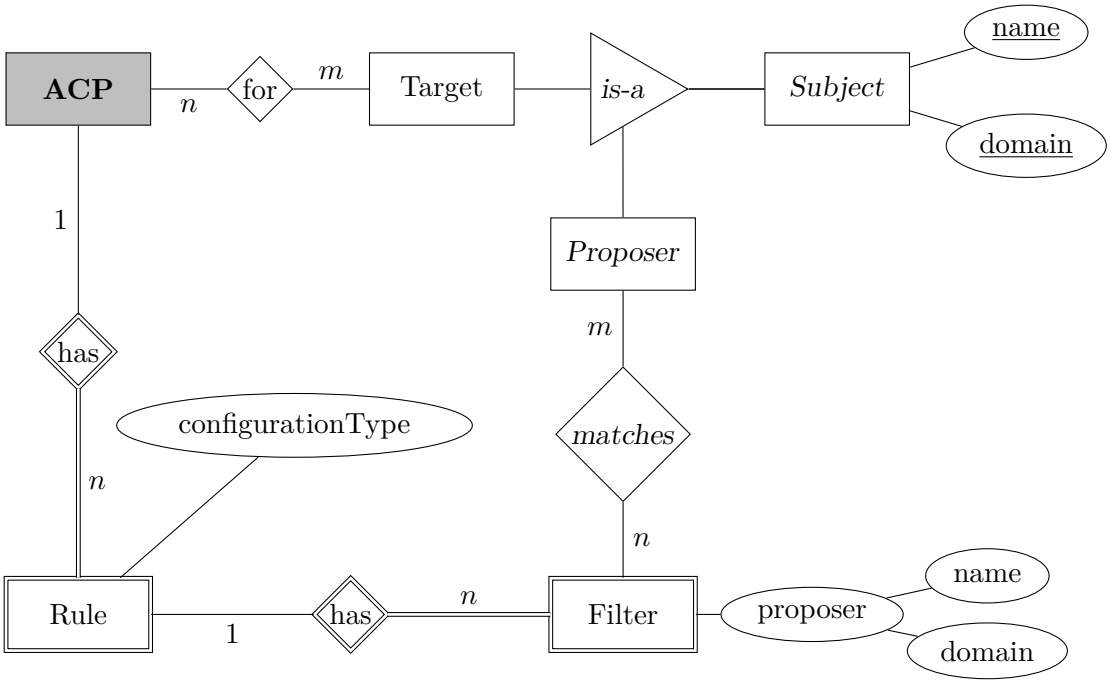


Figure 5.9: Structure of an ACP, illustrated with an ERM

```
{
  "targets": [
    {
      "name": "DeviceI",
      "domain": "Org1"
    }
  ],
  "rules": [
    {
      "configurationType": "ansiblePlaybook",
      "proposers": [
        {
          "name": "ProposerA",
          "domain": "Org1"
        },
        {
          "name": "ProposerB",
          "domain": "Org2"
        }
      ]
    }
  ]
}
```

Figure 5.10: Example of an ACP, represented in JSON, targeting a single device and entitling two dedicated proposers of different organizations

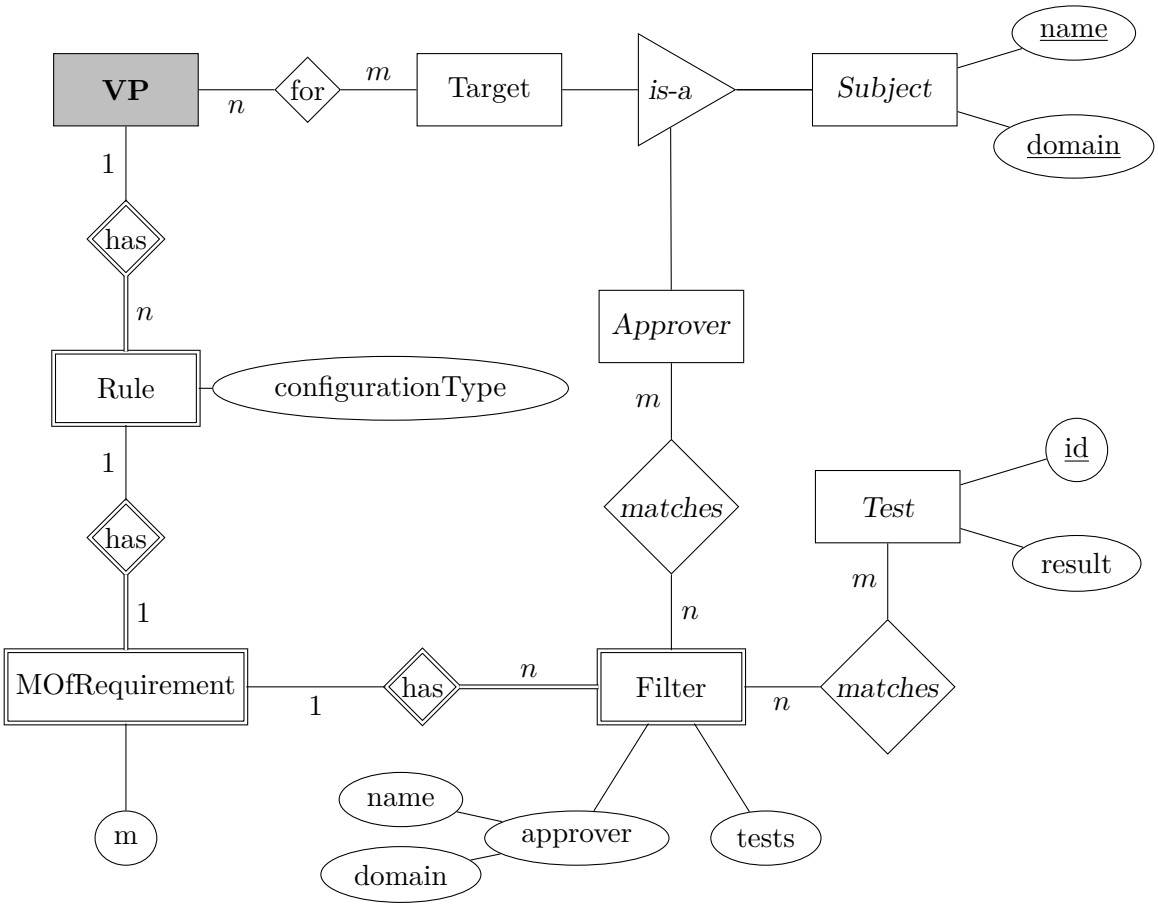


Figure 5.11: Structure of a VP, illustrated with an ERM

```
{
  "targets": [
    { "name": "DeviceI", "domain": "Org1" }
  ],
  "rules": [
    {
      "configurationType": "ansiblePlaybook",
      "mOfRequirement": {
        "m": 2,
        "filters": [
          {
            "approver": { "name": "ApproverA", "domain": "Org1" },
            "tests": [ { "id": "integrationTest", "result": "passed" } ]
          },
          {
            "approver": { "domain": "Org2" }
          }
        ]
      }
    }
  ]
}
```

Figure 5.12: Example of a VP, represented in JSON, targeting a single device and requiring two filters to be matched, one of them for a dedicated approver of *Org1* together with a specific test, and the other for any approver of *Org2*

5.5 3-Tier Architecture

We propose a 3-tier architecture with a presentation tier, a logic tier and a data tier, as depicted in figure 5.13. Each tier provides an interface for the next highest tier and can be deployed and implemented independently from the others. With this approach, our system can be operated with different infrastructure and in different environments without having to change the overall architecture.

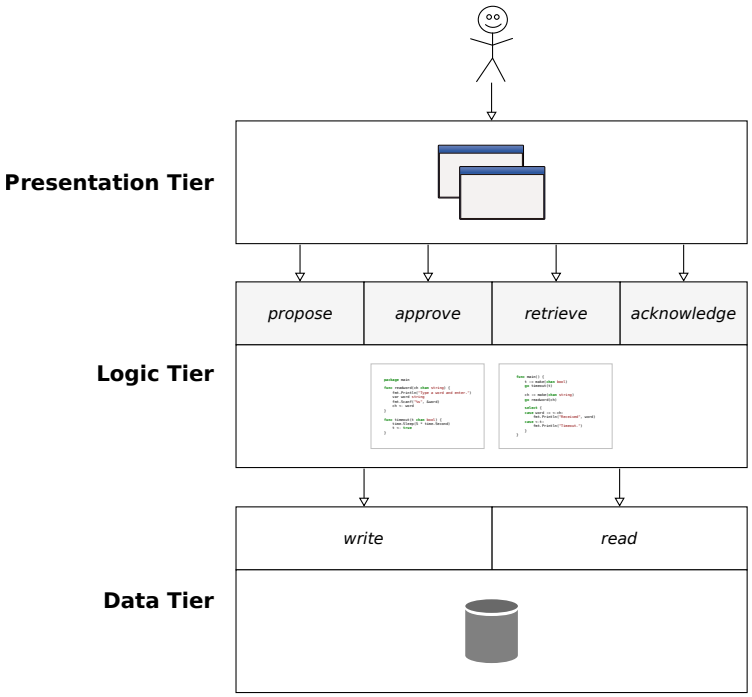


Figure 5.13: 3-tier architecture, consisting of the presentation tier, the logic tier and the data tier

The three tiers communicate over a well-defined protocol, which is based on the respective operations provided by the tiers. Figure 5.14 gives a high-level overview of that protocol in an UML sequence diagram, comprising the proposal, approval, retrieval and acknowledgement of a CR. In particular, it illustrates the characteristic of Multi-Party Authorization (MPA) introduced by our system: a target applies a configuration only after several approvers have reviewed and approved a configuration that had been proposed beforehand.

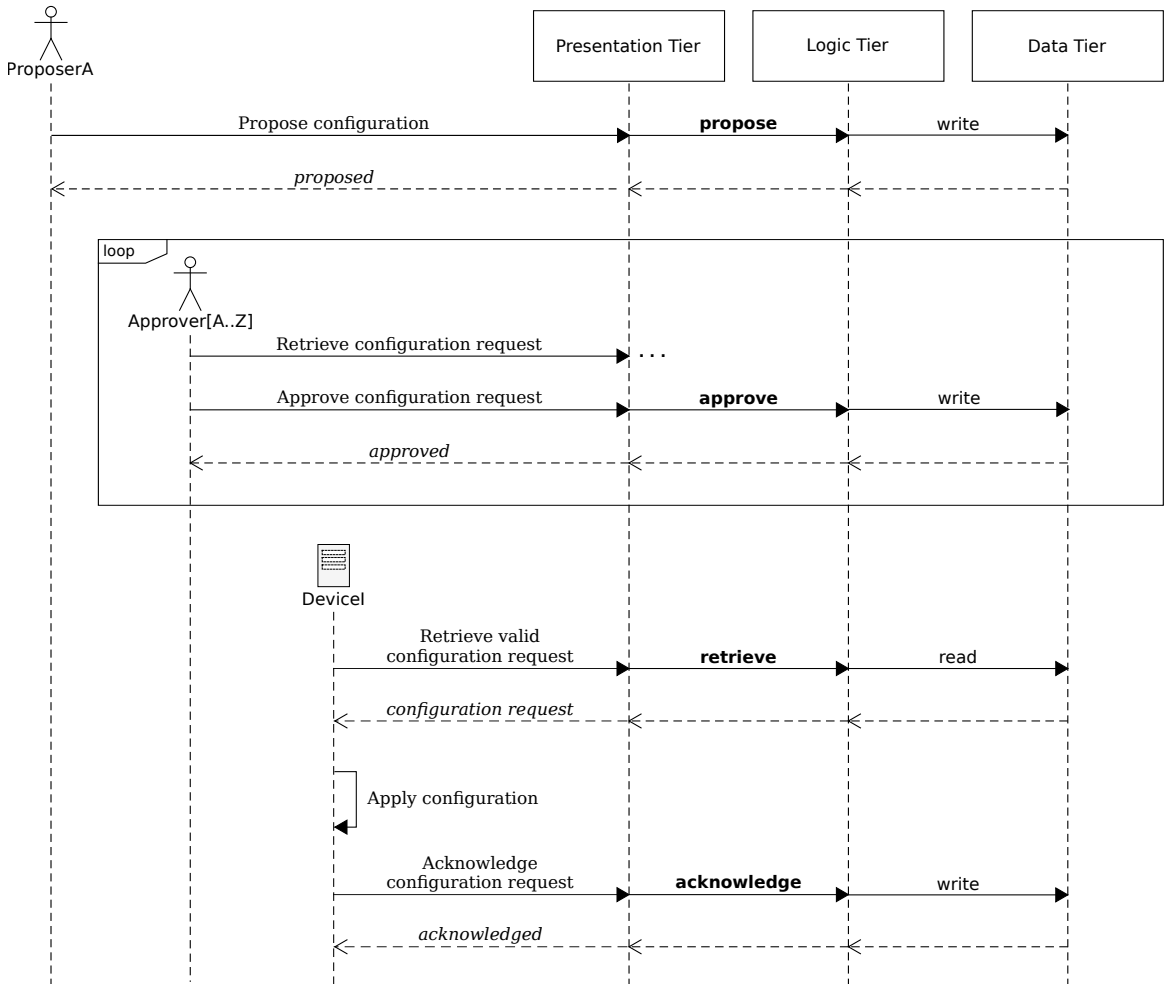


Figure 5.14: High-level overview of the 3-tier protocol, illustrated with an UML sequence diagram

5.5.1 Presentation Tier

The presentation tier serves as a user interface and usually runs on the clients. Different implementations for different purposes are possible here, including but not limited to

- a Command-Line Interface (CLI)
- a Representational State Transfer (REST) API
- a Graphical User Interface (GUI)

It communicates with the logic tier on behalf of the users and therefore enables them to perform the operations offered by it, namely to propose, approve, retrieve and acknowledge CRs. Additionally, the presentation tier may offer more complex operations based on the logic tier's operations, for instance by aggregating CRs or by calculating statistics.

5.5.2 Logic Tier

The logic tier handles the system's business logic and is accessed by the presentation tier. Its interface provides operations to propose and approve CRs on the hand and to retrieve and acknowledge them on the other hand. Therefore, it is responsible for managing the CRs. To persist the CRs, the logic tier accesses the data tier. Figure 5.15 shows the abstract and implementation-independent description of the logic tier's interface in the Interface Definition Language (IDL) [55].

```
1  interface Logic {
2
3    exception Error {};
4
5    CR propose(Configuration configuration, sequence<Target> targets)
6      raises(Error);
7
8    void approve(string crID, sequence<Test> tests)
9      raises(Error);
10
11   sequence<CR> retrieve(CRFilter filter)
12     raises(Error);
13
14   void acknowledge(string crID)
15     raises(Error);
16
17 }
```

Figure 5.15: Interface of the logic tier, described in the IDL, basing upon the custom data types defined in figure 5.6

The deployment of the logic tier depends on the infrastructure and the intended use case of the system. It can be deployed in a traditional way to an application server. Alternatively, it can be deployed to a distributed execution environment, like a peer-to-peer network with a smart contract engine.

5.5.3 Data Tier

The data tier is responsible for persisting the data of the system. Its two operations allow the next highest tier to *write* to and to *read* from the underlying storage, such as a distributed ledger.

It solely depends on the implementation of the data tier how the data storage is managed. If a traditional database management system was employed, the data objects would simply be written to and read from the database.

By basing the data tier upon a distributed ledger managed by a peer-to-peer network, however, we can ensure unforgeability, traceability and unerasability for write and read operations by design. Only after a ledger transaction has reached consensus in the network, the ledger state is updated accordingly. A single participant of the network is not able to forge a transaction. Furthermore, the blockchain-based transaction log, which is stored on the peer nodes and includes all transactions in a sequential order, ensures that all changes to the ledger state remain traceable and cannot be erased.

5.6 Conclusion

In this chapter, we have described the conceptual architecture of our system. It is designed to support different ways of deploying and implementing the 3 tiers. The centerpiece of this work is to utilize the characteristics of a blockchain-based distributed ledger for providing a configuration management system based on Multi-Party Authorization (MPA) and an unforgeable, traceable and unerasable data storage. The following chapter offers a concrete implementation for that purpose.

Chapter 6

Implementation

In this chapter, we present the implementation of TANCS. It follows the high-level design introduced in chapter 5 and serves as a proof of concept.

6.1 Overview

The implementation of TANCS is based on the 3-tier architecture described in chapter 5 and written in Go[1] 1.9[2]. The presentation tier runs on the client and offers a Command-Line Interface (CLI) that can be used by any actor which takes the role of a proposer, an approver or a target. The essential part in our implementation, however, plays Hyperledger Fabric. The logic as well as the data tier run in a Fabric network, composed of a set of peers and an ordering service. The whole business logic is executed as chaincode on the peers. Our implementation includes two chaincodes: the *management chaincode* (MGTCC) responsible for proposing, approving, retrieving and acknowledging a CR, and the *policy evaluation chaincode* (PECC) responsible for evaluating if an artifact fulfills an ACP and a VP, respectively. Finally, the data tier corresponds to the ledger managed by the peers, consisting of the blockchain and the state database.

It is crucial to understand that we utilize Hyperledger Fabric in two ways: on the one hand, the Fabric network serves as the distributed execution environment for the system's business logic. On the other hand, the distributed ledger of the Fabric network

[1] `https://golang.org/`

[2] `https://golang.org/doc/go1.9/`

serves as the repository for the system's data. With that approach, we can ensure unforgeability, traceability and unerasability for the operations as well as the data by design. Furthermore, we do not depend on a single trusted third party to run and administrate the system. With this in mind, it is even possible to manage configurations for infrastructure shared across stakeholders that only share a limited amount of trust, like a consortium of different organizations.

We choose Ansible as a configuration management framework since its configuration scripts, the so-called *playbooks*, are written in an easy to understand, declarative scripting language and support idempotence. A playbook is the actual payload of a CR and stored in its proposal. After a target has retrieved a valid CR from our system, the playbook is applied locally to that target. However, Ansible can be replaced by any other configuration management framework with very few adjustments, as our system does not depend on a specific configuration format.

Our implementation is available on LRZ GitLab[1]. A comprehensive overview of its package structure is given in appendix A.1.1. The whole implementation is licensed under the GNU Affero General Public License (AGPL) v3. Besides the copyleft principle, this license ensures that users have the right to obtain a copy of the software's source code even if it is operated as a remote service.

6.2 Deployment

Before being able to operate our system, a Fabric network with a set of peers and an ordering service has to be set up. Furthermore, the chaincodes that handle the system's business logic have to be installed on a subset of these peers, the endorsing peers. For conducting the setup and deployment process, our implementation includes a *Docker*[2] infrastructure and a purpose-built bootstrapper. Together, they provide an environment for testing and demonstration purposes than can be customized and extended easily.

On the one hand, the Docker infrastructure spans an exemplary Fabric network involving two organizations with one peer each. The supplied cryptographic material and network configuration have been generated with the official toolchain of the Fabric project, in particular using *cryptogen* and *configtxgen*. On the other hand, the bootstrapper is responsible for creating a new channel, joining the organizations' peers to that channel,

[1] `https://gitlab.lrz.de/tumi8/trustworthyconfigmanagement/`

[2] `https://www.docker.com/`

and finally installing and instantiating our chaincodes. It is designed to be idempotent. Therefore, it can be re-run on an already bootstrapped network, being able to detect only the changes that have to be applied. This is especially useful for joining new organizations or peers to an existing network and for upgrading chaincodes installed on peers to a new version.

With this setup, it is possible to pursue a configuration management process across the boundaries of different stakeholders. This typically involves an endorsement policy that requires endorsements from peers of all participating stakeholders. If, for instance, the system manages configurations for targets that belong to different organizational units, the endorsement policies may specify that a transaction has to be endorsed by at least one peer of each organizational unit to prevent one unit from acting maliciously by manipulating execution results.

Table 6.1 gives an overview of the exemplary Fabric network, covering the entities and the corresponding Docker services. We employ a *solo* ordering service that is suitable for testing purposes only, as it is a single point of failure. The peers of the organizations *Org1* and *Org2* endorse transactions for the chaincodes MGTCC and PECC installed on them. To enable rich queries with respect to the ledger data, the state databases for both peers are managed by the document-oriented database software *CouchDB*[1]. A dedicated Docker service is provided for the bootstrapper. As soon as this service is started, the bootstrapper begins to initialize the network. Clients can access the network via the CLI service, which comes with an installation of the CLI together with exemplary client identities.

Entity	**Unit**	**Responsibility**	**Docker**	
			Service	**Image**
Orderer	OrdererOrg	Solely orders transactions	`orderer1`	`fabric-orderer`
Peer0	Org1	Endorses transactions	`org1peer0` `org1peer0couchdb`	`fabric-peer` `fabric-couchdb`
Peer0	Org2	Endorses transactions	`org2peer0` `org2peer0couchdb`	`fabric-peer` `fabric-couchdb`
Bootstrapper		Bootstraps the network	`bootstrapper`	`fabric-baseimage`
Client		Runs the CLI	`cli`	`fabric-baseimage`

Table 6.1: Entities and Docker services of the exemplary Fabric network

[1] `https://couchdb.apache.org/`

6.3 Tiers

In the following, we explain the interactions of the different tiers in our implementation and illustrate them with UML sequence diagrams. Later on, we describe the implementation of each tier in detail.

6.3.1 Interactions

At the beginning of each configuration management workflow, a proposer proposes a configuration via the CLI, as depicted by step 1.1 in figure 6.1. The command's arguments are the targets and the configuration data itself, for instance in the form of an Ansible playbook. The CLI parses and interprets the command and calls the logic tier by invoking the *propose* operation of the MGTCC (1.2). The MGTCC first checks if the proposer is permitted to propose a configuration for the specified targets according to the corresponding ACP. For this, it calls the *evaluateACP* operation of the PECC (1.3). If the PECC returns a positive result, a new CR will be created and put into the ledger (1.4).

The chaincode invocation in step 1.2 leads to a transaction flow in the Fabric network that is not shown in the figure, but has been described in figure 2.6 of section 2.3.3. Here, the proposer serves as the client in the Fabric network and sends a transaction proposal to the endorsing peers, which simulate the *propose* operation and return their respective endorsement to the client. Afterwards, the client sends the transaction together with the endorsements to the ordering service, which will eventually broadcast a new block to all peers of the network to be appended to their blockchain copies. The peers check the transaction's validity and, if fulfilled, finally write the CR contained in the transaction to their copy of the state database. Subsequently, the proposer is notified that the CR has been successfully proposed.

Figure 6.2 shows the approval of a CR: after the approvers have noticed a new pending CR, either by manually retrieving it from the system (2.1) or by getting notified dynamically, they review the CR. If they agree to the configuration change, they call the CLI to approve the CR (2.2). The CLI again parses and interprets the command and hands it over to the logic tier to invoke the *approve* operation of the MGTCC (2.3). The MGTCC adds the new approval to the set of approvals contained in the CR. Then, it calls the *evaluateVP* operation of the PECC to check if the CR is already valid according to the corresponding VP (2.4). If the PECC returns a positive result, the MGTCC will change the CR's status to *valid*. Then, it puts the modified CR into the ledger (2.5). The transaction flow in the Fabric network is analogous to the invocation

of the *propose* operation. Eventually, the approver will be notified that the approval succeeded.

To retrieve CRs matching a set of specified criteria, a subject calls the respective CLI operation, as depicted by step 3.1 in 6.3. The CLI again calls the logic tier (3.2). Since retrieving a CR does not modify it, it is sufficient to perform a *query* using the *retrieve* operation of the MGTCC that actually gets the CR from the ledger (3.3). This time, the transaction flow just involves the endorsing peers which simulate the *retrieve* operation and return the signed query results to the client. Then, the client verifies that the results match. Usually, queries do not lead to transactions sent to the ordering service and committed to the peers.

Finally, figure 6.4 shows the acknowledgement of a CR: after having retrieved a valid CR (4.1), the target applies the configuration locally (4.2), for instance by utilizing a configuration management framework like Ansible. In case of success, the target acknowledges the CR (4.3), followed by the call of the logic tier (4.4). The *acknowledge* operation of the MGTCC adds a new acknowledgement to the CR's set of acknowledgements. After each of the targets has acknowledged the CR, the MGTCC changes the CR's state to *acknowledged*. Then, it puts the modified CR into the ledger (4.5).

6.3.2 Presentation Tier

The presentation tier is implemented in Go and provides a Command-Line Interface (CLI) compatible to the GNU Coding Standards [25]. The well-maintained library *kingpin*[1] by Alec Thomas is used for parsing the command-line arguments and flags.

Extensive documentation and usage instructions are offered by the built-in *help* command. The excerpt in table 6.2 gives an overview of the commands, arguments and flags of the CLI. All configuration management operations are bundled into a single program, simplifying the deployment to the different users like proposers, approvers and targets.

For every operation, a set of general flags must be specified. On the one hand, it is required to name an `identity-provider` with optional initialization parameters. The CLI is able to deal with arbitrary identity providers that implement a particular interface. Currently, it supports Hyperledger Fabric as an identity provider. The given `identity`, in combination with the appropriate `identity-authentication` data such as a password or a key, is then authenticated with respect to the given identity provider.

[1] `https://gopkg.in/alecthomas/kingpin.v2`

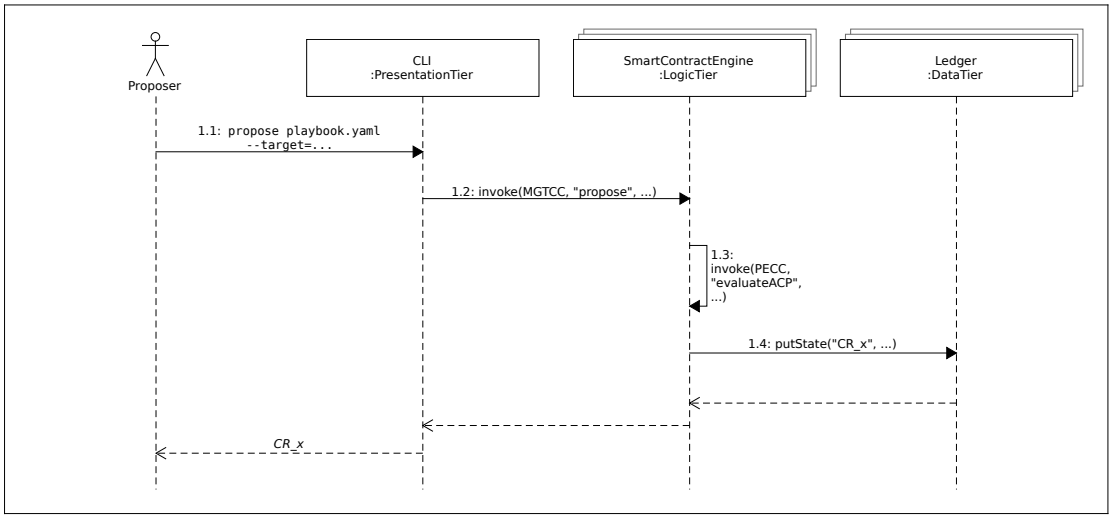


Figure 6.1: Tier interactions for proposing a configuration, illustrated with an UML sequence diagram

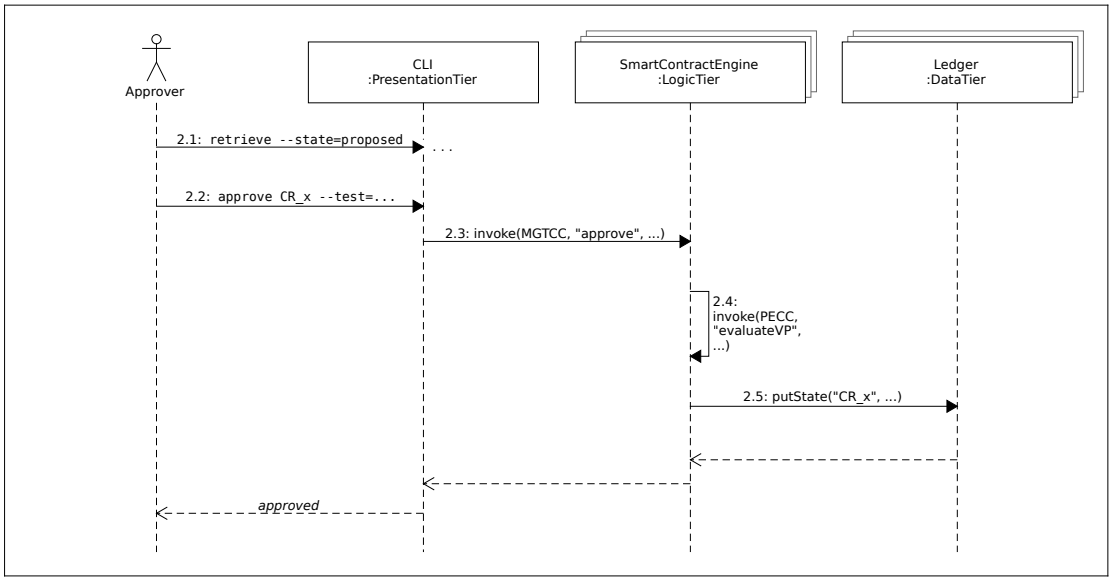


Figure 6.2: Tier interactions for approving a CR, illustrated with an UML sequence diagram

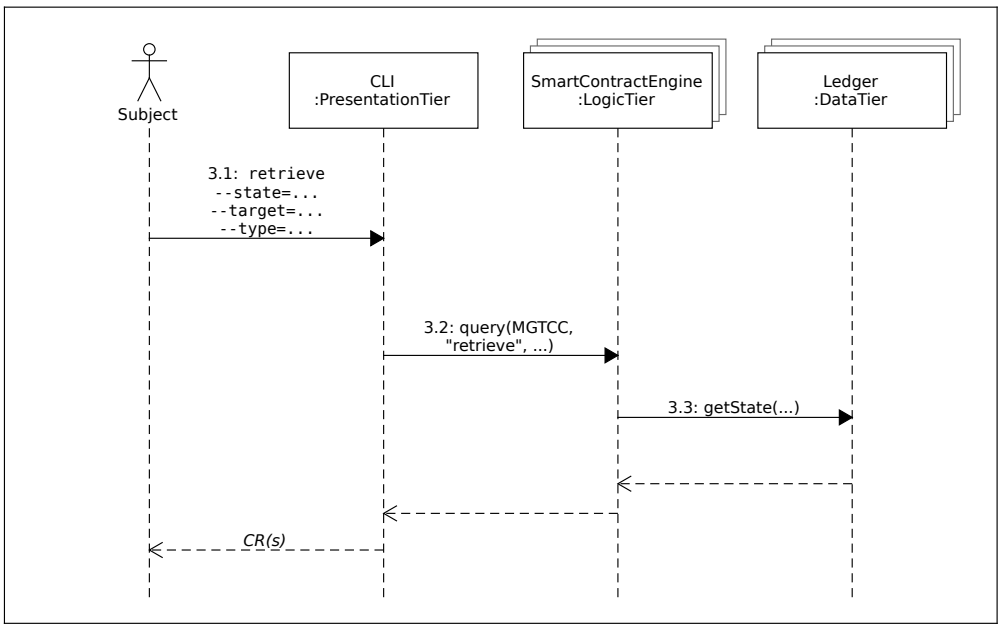


Figure 6.3: Tier interactions for retrieving CRs, illustrated with an UML sequence diagram

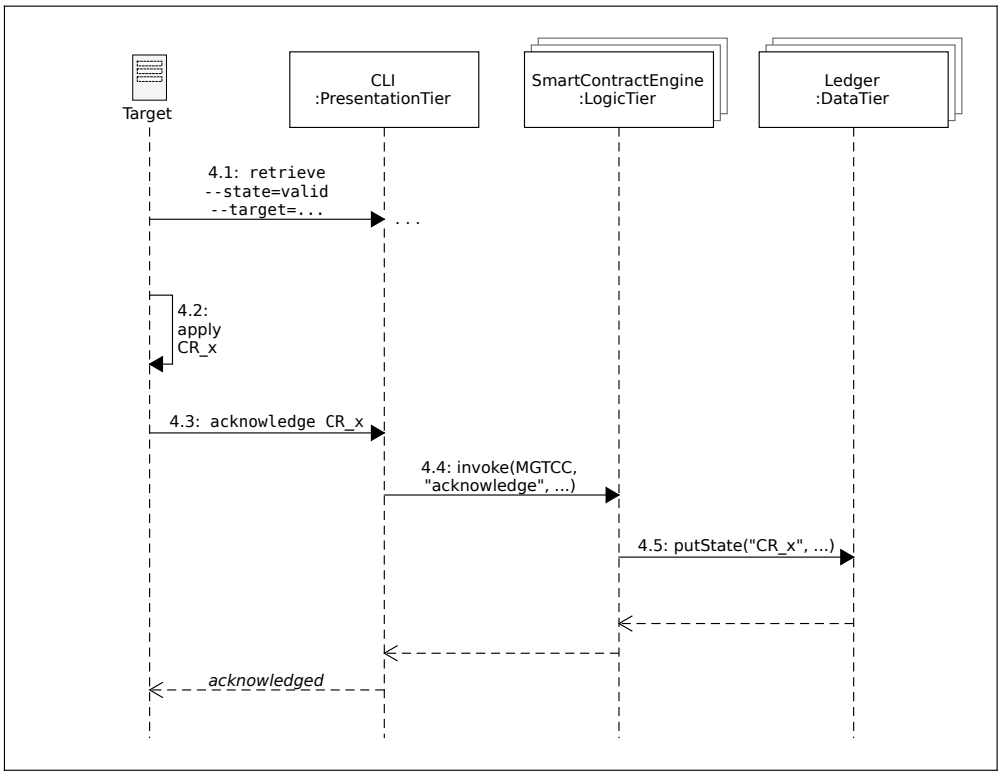


Figure 6.4: Tier interactions for acknowledging a CR, illustrated with an UML sequence diagram

On the other hand, it is required to specify a `client`, again with optional initialization parameters. A client is the software component that actually implements the business logic. Since our architecture does not depend on a specific implementation of the business logic, the CLI can communicate with arbitrary clients via a well-defined interface. At the time of writing this thesis, a client implementation for Hyperledger Fabric based on chaincodes is available.

The CLI allows users to execute the configuration management operations `propose`, `approve`, `retrieve` and `acknowledge`, as introduced in chapter 5. Moreover, it provides a `subscribe` operation, which runs as a daemon and outputs occurring CRs that match the specified criteria, like state and targets. Internally, the event architecture of Fabric is used to get notified of new CRs. For each subscription, a `handler` can be specified to handle the matching CRs. A handler is an executable program that is passed a matching CR and optional arguments. The handling procedure can be influenced in several ways: it is possible to pass the entire CR or just its configuration to the handler, either as a file path argument or via the standard input. If the `auto-acknowledge` mode is enabled, successfully handled CRs will be automatically acknowledged to the system.

A concrete example for a handler is the command-line tool `ansible-playbook` which is part of the Ansible suite. It runs a given playbook locally. For that, the tool expects a path to a locally stored playbook file. Moreover, it supports several optional parameters that influence its behaviour. Together with a subscription to valid CRs addressed to itself, a target is then able to run the contained playbooks automatically, and additionally submit acknowledgements in the `auto-acknowledge` mode.

6.3.3 Logic Tier

For the logic tier, we provide an implementation that conducts the configuration management process in a tamper-resistant way by leveraging Hyperledger Fabric as a *smart contract engine*. The tier is accessed by the CLI or any other implementation of the presentation tier.

API and Components

Our API consists of two Go-specific interfaces: the `Client` interface represents the actual business logic and supports the configuration management operations *propose*, *approve*, *retrieve* and *acknowledge*. It is based on the abstract IDL interface introduced in section 5.5.2 and is directly accessed by the presentation tier. The `IdentityProvider` interface

Commands	Arguments	Flags
		identity *name@domain* identity-authentication ... *string* identity-provider fabric identity-provider-initialization ... *string* client fabric client-initialization ... *string*
help		
propose	*configuration file*	target ... *name@domain* ... type acp|vp|ansiblePlaybook
approve	*id string*	test ... *id:result*
retrieve		id *string* state proposed|valid|acknowledged|outdated target ... *name@domain* ... type acp|vp|ansiblePlaybook print-no-header *bool* print-null *bool* print-raw-id *bool* print-raw-configuration-data *bool*
acknowledge	*id string*	
subscribe		id *string* state proposed|valid|acknowledged|outdated target ... *name@domain* ... type acp|vp|ansiblePlaybook handler *file* handler-argument ... *string* pass-entire-cr *bool* pass-via-stdin *bool* auto-acknowledge *bool* exit-on-error *bool*

Table 6.2: Excerpt of the CLI's commands, arguments and flags

serves as a communication endpoint to the employed identity provider and supports a single operation, namely to *authenticate* a given subject.

We offer concrete implementations of these interfaces based on Hyperledger Fabric 1.1.0[1]: the component `fabricClient` is an implementation of `Client` and executes the configuration management operations by invoking chaincodes in a Fabric network, while the component `fabricIdentityProvider` is an implementation of `IdentityProvider` and employs the PKI-based user identities that are defined in the configuration of a Fabric network.

[1] `https://github.com/hyperledger/fabric/releases/tag/v1.1.0/`

The communication between a client and the actual nodes of the Fabric network, in particular the peer and orderer nodes, is carried out by *fabric-sdk-go*[1], the official Software Development Kit (SDK) for Hyperledger Fabric and Go. At the time of writing this thesis, a stable release is not yet available. However, the developers offer an alpha version that we employ in our implementation. The UML component diagram in figure 6.5 illustrates the relationships between `fabricClient`, `fabricIdentityProvider` and the SDK.

We make use of the two most elementary operations of the SDK: *invoke* triggers a chaincode invocation in a Fabric network with the aim to modify the ledger, while *query* only triggers a chaincode invocation for reading from the ledger, without involving a transaction to be committed to the peers. Although a read operation does not modify the ledger, it is yet possible to process it as an ordinary transaction using *invoke*. With that approach, the ledger additionally contains a history of read operations. However, this may have negative effects on the overall performance due to the high number of read accesses compared to write accesses.

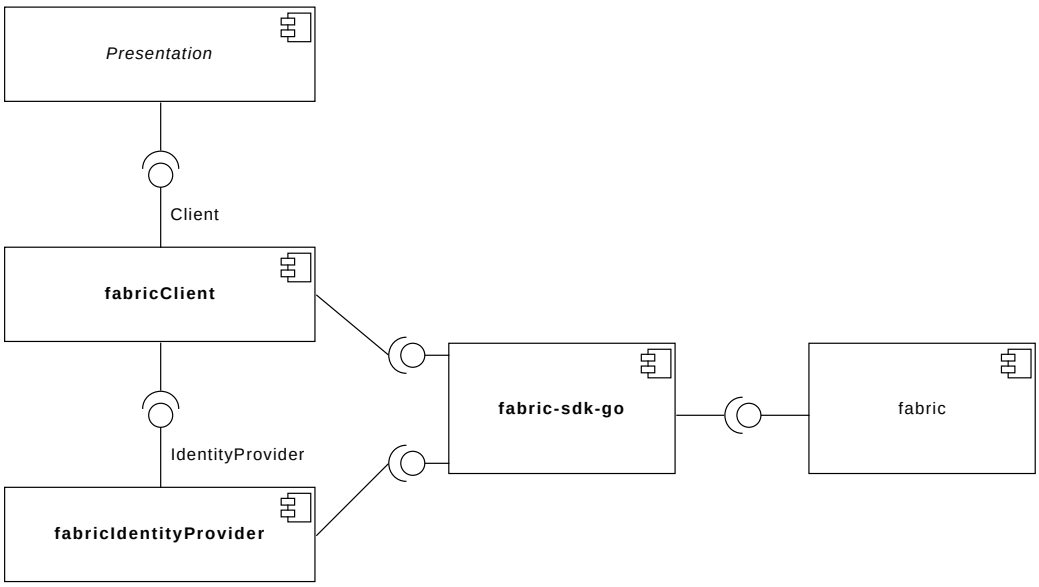


Figure 6.5: Component dependencies on the logic tier, illustrated with an UML component diagram

[1] `https://gopkg.in/hyperledger/fabric-sdk-go.v0`

Chaincodes

The implementation of the logic tier is split into two independent chaincodes, both written in Go: on the one hand, the *management chaincode* (MGTCC) implements the operations *propose*, *approve*, *retrieve* and *acknowledge*. Each of these operations conforms to the respective process model introduced in section 5.2.

On the other hand, the *policy evaluation chaincode* (PECC) implements the operations *evaluateACP* and *evaluateVP*. The first operation determines if a proposer is permitted to propose a configuration for a set of targets according to a given ACP, while the latter determines if a CR is valid according to a given VP. For that, the PECC is invoked by the MGTCC dynamically at runtime.

6.3.4 Data Tier

The data tier is represented by the ledger of a Fabric network itself, consisting of the actual blockchain and the state database. The data tier's abstract operations introduced in section 5.5 refer to the two elementary operations of Fabric's interface `shim.ChaincodeStubInterface` [48]: while *write* is implemented by the operation *PutState*, *read* is implemented by the operation *GetState*.

PutState is aimed to store the given key-value pair in the ledger. For that, it adds the pair to the writeset of the transaction proposal. In contrast, *GetState* is aimed to read a key-value pair from the ledger. Consequently, the key is added to the readset of the transaction proposal. As described in section 2.3.3, modifications to the ledger only take place after the transaction proposal with the read/writeset has been properly endorsed, processed by the ordering service and finally validated by the peers.

Chapter 7

Evaluation

This chapter evaluates the interoperability, performance and security of TANCS, considering its high-level design as well as its implementation based on Hyperledger Fabric. For that purpose, the requirements defined in chapter 3 serve as a reference.

7.1 Interoperability

To effectively make use of a new system, it must be able to interoperate with the existing systems of an IT landscape. In our case, this involves the management and authentication of identities, the specification and evaluation of policies, and the communication with established configuration management frameworks.

7.1.1 Identity Management

Both the design and implementation of our system provide the possibility to communicate with an arbitrary identity provider via a well-defined interface of the logic tier. Essentially, the interface consists of an *authenticate* method that expects the identifier of a subject together with generic authentication data, for instance represented by a password or a private key. Protocols for challenge-response authentication are not yet supported, but may be added in the future.

Different implementations for that interface are possible, for instance one based on an LDAP-compatible directory service operated by an organization. Since our goal is to conduct a configuration management process based on Multi-Party Authorization (MPA) in a tamper-resistant way, however, it is crucial to ensure that the process

cannot be compromised due to a weakly secured and easily attackable identity provider. Typical examples include identity providers that are fully centralized, or those that are comprehensively accessible by administrators whose rights our system aims to restrain. That is why all participating stakeholders, for instance a consortium of organizations, have to control and operate their own identity management in a secured way.

However, there are alternative approaches to traditional identity management. The field of decentralized identity management is in the spotlight of current research and development, with an overview of existing solutions given by Dunphy and Petitcolas [21]. First approaches in this field have been taken with Namecoin [33] in 2011 and Blockstack [1] in 2013. The basis for a decentralized name system is typically provided by a public peer-to-peer network which is large enough not to be taken over easily by attackers. The name system itself can be built on top of such a network as a virtual chain together with a state machine for domain names, with the different states representing unregistered, registered and revoked names. Internally, names and namespaces can be stored in zone files similar to those of DNS. To register a new name, users follow a dedicated registration protocol which may include the payment of a registration fee to the creator of the next block. Current research deals with solving a range of efficiency and scalability issues of these approaches, for instance concerning the system's caching behaviour and the amount of data to be stored by a node.

7.1.2 Policy Evaluation

Our implementation comes with a basic policy language as well as a basic implementation of a policy evaluator compatible to it. In the future, further use cases may arise for our system, potentially requiring the support of a more powerful policy language and evaluator. That is why our system accesses the policy evaluator in use via a well-defined interface, making it possible to easily employ a different implementation, which in turn may be based on an existing and comprehensive policy framework, such as the Policy Machine presented by the National Institute of Standards and Technology (NIST) as part of their standardization project Next Generation Access Control (NGAC) [23].

7.1.3 Configuration Management

For applying configurations to targets, our system is able to conveniently interoperate with configuration management frameworks such as Ansible, Puppet or Chef. The communication with those frameworks takes place on the presentation tier of our system, particularly in the implementation of the Command-Line Interface (CLI): technically

based on the event architecture of Hyperledger Fabric, it allows clients to *subscribe* to incoming CRs matching a specified filter.

Fabric's event architecture provides an efficient and flexible way for clients to get notified if new blocks are added to the ledger, specific transactions are committed, or custom chaincode events are triggered during the execution of a chaincode. In our implementation, the *management chaincode* (MGTCC) produces a chaincode event for each state update of each CR. Clients can then subscribe to that kind of event.

If an incoming CR matches the specified filter, the presentation tier will hand it over to a specified *handler*, which basically is an arbitrary executable program. All configuration management frameworks introduced in section 2.1 provide stand-alone tools for applying their configuration scripts locally to targets. Table 7.1 gives an overview of the available tools, each of them officially maintained by the framework's development team and compatible with the *handler* interface of our system.

Framework	**Tool**	**Description**
Ansible	`ansible-playbook`	*"Runs Ansible **playbooks**, executing the defined tasks on the targeted hosts"* [4]
Puppet	`puppet apply`	*"Applies a standalone Puppet **manifest** to the local system"* [59]
Chef	`chef-solo`	*"Executes chef-client in a way that does not require the Chef server in order to converge **cookbooks**"* [18]

Table 7.1: Overview of stand-alone tools for applying configuration scripts, compatible with the *handler* interface of our system

By employing our system as the exclusive management and storage component for configurations and by narrowing the purpose of the presented configuration management frameworks to just apply configuration scripts to targets, many of their functionalities are not utilized any more. That is why other ways of interoperating with these frameworks might be worth considering, such as synchronizing the valid configurations stored in our system with the configuration repository offered by some frameworks. Moreover, several frameworks include comprehensive features for managing the device infrastructure itself. These capabilities might be leveraged in our system, for instance to provide the identities of targets.

7.1.4 Reusability

As stated in chapter 5, the design of our system is reusable for different implementations and environments. In particular, the 3-tier architecture allows to exchange and adapt the implementation of a tier independently from the others. Consequently, other distributed ledger frameworks than Hyperledger Fabric can be employed, like Hyperledger Sawtooth. The application logic, currently implemented as Fabric chaincode, and the bootstrapper must then be adapted to the architecture of the other framework. At the end of the day, however, major parts of our implementation can be reused.

If the intended distributed ledger framework provides no execution environment, for instance in the form of a smart contract engine, the application logic, residing on the logic tier, will have to be executed on the clients. Then, the framework will only operate on the data tier, providing a distributed and tamper-resistant data storage. MultiChain, as introduced in section 2.3, is an example for such a framework.

Deploying our system to a public and unpermissioned distributed ledger framework like Ethereum is a much more challenging task. Typically, this approach only has benefits if the configuration of a certain infrastructure has to be justified to the public. Our application logic would have to be translated to an Ethereum smart contract written in Solidity. Since the authorization process of our system is based on authenticated identities, each identity would have to be linked to an Ethereum account. Alternatively, the authorization process could be altered to an anonymous voting process. With respect to confidentiality, it may not be desirable to publish security-relevant configurations in a public ledger, hence requiring the encryption or hashing of the data. Lastly, it would have to be clarified how the payment of the Ethereum transaction fees can be organized.

7.2 Performance

Configuration management is strongly interconnected with the fields of software development and operation. Thus, it needs to be responsive and to steadily provide feedback to users and associated software systems. In the following, we describe factors and solutions that affect the responsiveness and performance of our system.

7.2.1 Latency by Multi-Party Authorization

To enforce Multi-Party Authorization (MPA) with respect to a proposed configuration, we conduct a configuration management process consisting of several steps and requiring actions by different subjects. Since different subjects specified in the targets' *validity*

policy (VP) have to approve a configuration, there is no upper time limit for a configuration to become valid. The same holds for the targets to apply and acknowledge a valid configuration, although this task can be fully automated.

Therefore, an arbitrarily long period of time can pass between the proposal of a configuration and its activation on the intended targets. Depending on the use case of our system, it may be necessary to reduce or otherwise influence this kind of response time, for instance by introducing an expiry mechanism for CRs. If a CR did not become valid within a specified period of time, its state would be set to *outdated* automatically, with no possibility to modify that particular CR ever again. Of course, this measure does not necessarily lead to an accelerated approval process, but provokes a new state for a pending CR that can be reacted to, for instance by adapting the involved policies or by proposing a different configuration.

7.2.2 Latency by Distributed Ledger Technology

Another kind of latency is induced by the use of Distributed Ledger Technology (DLT) itself. Our system's execution and storage model leverages the distributed and decentralized character of a peer-to-peer network, introducing a significant overhead regarding the communication, coordination and synchronization within in the network. In particular, operations are executed redundantly by design to prevent participants from acting maliciously. Compared to DLT frameworks based on time-consuming consensus mechanisms like Proof of Work (PoW), however, the latency induced by the consensus mechanism of Hyperledger Fabric is negligible.

In contrast to the response time caused by our MPA-based process, the latency of DLT can be easily controlled on a technical level. Increasing the capacity and efficiency of the participants' hardware and software can already lead to a significant reduction. This involves the employment of up-to-date CPUs and sufficient amounts of memory for the endorsing peers of a Fabric network, but also slim and scalable protocols for their communication with each other.

Storage Model

In our current implementation, the actual configuration data is stored in the ledger directly. This allows to guarantee unerasability and unforgeability of the configuration data out of the box. However, all data records are stored redundantly on each participating peer. This might be unproblematic for a small amount of configuration data, but can quickly lead to serious performance and capacity issues once the amount increases.

As an alternative, the ledger may only store a URI to the configuration data together with a hash value of the data. An external storage provider, such as a cloud service, would then be responsible for storing the configuration data permanently and with a focus on high availability. Every time the data would be retrieved from the external provider, its integrity would have to be checked using the hash value stored in the ledger.

Ledger Pruning

Independent of the concrete storage model, there may be the need to prune the ledger. Since the ledger constantly grows, the required disk space increases. This is especially problematic for peers hosted on machines with a limited amount of disk space, like single-board computers or embedded systems. Planned to be implemented in future versions of Hyperledger Fabric, the developers describe an approach that allows to maintain an abstraction of the ledger's blockchain containing valid transactions only [37]. Invalid transactions are omitted, as they are irrelevant for the ledger state.

To increase the performance and to decrease the required disk space, further kinds of abstractions are imaginable. A typical example would be a peer that does not store a full ledger copy starting with the genesis block itself. Instead, its copy would start with a successive block that has been verified to serve as a trustworthy starting point.

State Database

As introduced in section 2.3.3, a ledger in Hyperledger Fabric consists of the actual blockchain and a state database. The latter represents the current state of the ledger, which results from applying the valid transactions of the chain's blocks in consecutive order. From a conceptual point of view, the state database is not necessary, but it provides means to efficiently query the ledger. Generally, Fabric's architecture allows to conveniently exchange the database implementation in use. We choose CouchDB[1], since it currently is the only supported database being able to cope with *rich queries*. With these type of queries, it is not only possible to retrieve data records by their keys, but by a set of arbitrary attributes.

The database architecture of CouchDB is document-oriented, with its query model based on *MapReduce* [19], a framework for efficiently processing big amounts of data in a distributed and parallelized manner. CouchDB allows its users to define the *map* and *reduce* routines in so-called *views*. At the time of writing this thesis, however, Fabric

[1] `https://couchdb.apache.org/`

offers no integrated feature to manage and access those views. As an alternative, Fabric provides integrated support for CouchDB's ad-hoc query functionality. It is based on the *Mango* query language[1] with its declarative JSON syntax, that in turn was inspired by MongoDB. CouchDB provides a Representational State Transfer (REST) API to perform such queries [7]. To optimize the query time, the API also offers the possibility to create custom *indexes*. Internally, those indexes are again based on views. Index definitions can be shipped with every Fabric chaincode, leading to an automatic creation of the indexes once the chaincode is instantiated.

Internally, CouchDB implements Multi-Version Concurrency Control (MVCC) by employing an *append-only* storage model [9]. New versions of documents are appended to an ever-growing B^+ tree, without the old ones being deleted. With that approach, no locking system has to be introduced for allowing simultaneous read and write operations, potentially leading to a better performance. However, this requires more disk space, especially when storing large amounts of data that gets changed regularly. That is why CouchDB offers a *compaction* mechanism which compresses the database by discarding old revisions of documents [8].

7.3 Security

There is a range of factors affecting the security of our system. In the following, we evaluate our system with respect to the fulfillment of the security requirements defined in chapter 3, discuss the role of policies, and provide suggestions for managing software vulnerabilities and auditing processes.

7.3.1 Fulfillment of Security Requirements

The security requirements defined in section 3.2 are fulfilled in complementary ways on different levels of our system. We differentiate between two levels: on the one hand, we assess the application logic, which includes the logic executed on the clients as well as the chaincode logic executed on the peers. On the other hand, we consider Hyperledger Fabric itself, taking into account its architecture together with the security properties it provides. Table 7.2 gives an overview of how the different requirements are fulfilled on these two levels.

[1] `https://github.com/cloudant/mango/`

First of all, our application logic is responsible for performing the authentication. Every subject participating in our system has to be authenticated beforehand via an identity provider. In our implementation, we utilize the identities and authentication mechanisms provided by the Fabric network in use.

The application logic takes also part in providing accountability and traceability. Every configuration management operation, like proposing or approving, is associated with the subject in charge and stored in the corresponding CR together with a timestamp, resulting in a history of operations. On a technical level, Fabric is responsible for generating and verifying the cryptographic signatures of the corresponding transactions and for providing a traceable data storage.

Multi-Party Authorization (MPA) for configurations is again handled by our application logic. A target-dependent *validity policy* (VP) specifies requirements for configurations to become valid. Requiring m of n approvals for a configuration to become valid, with $n > m > 1$, prevents a single administrator from validating a configuration and therefore from causing damage. Fabric is not involved in the actual MPA process, but handles the corresponding transactions.

Finally, tamper-resistance, particularly including protected execution as well as unerasable and unforgeable data storage, is provided by Fabric. This involves the execution of chaincode on multiple peers according to the specified endorsement policy that requires k of l endorsements, again with $l > k > 1$. With that approach, no single peer is able to forge an execution result. Furthermore, Fabric is responsible for ordering the endorsed transactions. The employment of a single orderer node is suitable for testing purposes only, as the compromisation of this single node leads to a compromisation of the whole system. Instead, the ordering service has to consist of several orderer nodes that communicate via a Byzantine Fault Tolerance (BFT) protocol. Consequently, less than one third of the orderer nodes can act maliciously without affecting the functioning of the system. At the time of writing this thesis, there is only a proof-of-concept implementation of a BFT ordering service available [62], but an official one for productive use is under development.

Generally speaking, both the number of peers and orderer nodes influence the security of the overall system: the less nodes there are in the Fabric network, the easier it becomes for an attacker to compromise the system.

Requirement	**Level**	**Means**	**Instantiation**
Authentication	AL	Identity Provider	Utilization of Fabric client identities
Accountability	AL	Association of each operation with subject in charge	Recording of proposer, approver, acknowledger
	FAB	Cryptography	Generation and verification of signatures
Traceability	AL	History of operations for each CR	Recording of configuration proposal, list of approvals, list of acknowledgements
	FAB	Traceable and unforgeable storage	Distributed ledger
Multi-Party Authorization (MPA) of configurations	AL	Enforcement of *validity policy* (VP) stipulating multiple parties	VP requiring m of n approvals
Tamper-Resistance	FAB	Endorsement of transactions by multiple peers	Endorsement policy requiring k of l endorsements
		Ordering of transactions by multiple orderers	Ordering service guaranteeing Byzantine Fault Tolerance (BFT)

Table 7.2: Fulfillment of the security requirements on different levels, differentiating between Application Logic (**AL**) and Hyperledger Fabric (**FAB**)

7.3.2 Policy Model

In our system, decisions with respect to the proposal and approval of configurations are taken on the basis of Attribute-Based Access Control (ABAC). For that, an *access control policy* (ACP) or a *validity policy* (VP) has to be evaluated against a given artifact, such as a new configuration or a new approval. Defining requirements for a policy language and a policy evaluator on the one hand and integrating them into a system on the other hand is a non-trivial and security-relevant task. Vulnerabilities and uncontrollable complexity regarding policies and their evaluation can quickly break essential security properties such as authenticity, accountability and controlled access, as Jebbaoui et al. showcase in their study on the eXtensible Access Control Markup Language (XACML) [31].

Therefore, our system uses a policy language and policy evaluator that both are designed and implemented in a reductionistic and minimalist way. The reduced complexity allows us to guarantee the properties and requirements defined in chapter 3 and to test the evaluation routines comprehensively. Nevertheless, the users themselves are responsible for specifying policies that suit their security requirements. Therefore, special care has to be taken to avoid the specification of unsuitable and weak policies.

To conduct the actual evaluation of the policies, namely the ACPs and VPs, our Fabric-based implementation comes with a separate chaincode, the *policy evaluation chaincode* (PECC). This ensures separation of concerns, making it possible to deploy the PECC to peers dedicatedly responsible for policy evaluation, to apply stricter *endorsement policies*, and even to conduct the policy evaluation in a separate Fabric channel for confidentiality reasons. As mentioned in section 7.1.2, the implementation of the evaluator itself can be extended and exchanged conveniently. That way, even approaches providing means to prove the correctness of security policies can be employed, such as the formalization approach outlined by Battiato et al. [10].

7.3.3 Vulnerability Management

The way of dealing with software vulnerabilities can have an essential impact on a system's security. In particular, developers and operators of a system have to keep in mind that no software is perfect and vulnerabilities can be revealed at any time.

Hyperledger Fabric provides integrated mechanisms for managing the *lifecycle* of a chaincode. If vulnerabilities in one of our chaincodes are found, these mechanisms will allow to deploy an updated version of the chaincode to the network's peers and to perform the corresponding upgrade transaction. That way, flaws in chaincodes can be eliminated in an effective and efficient way. Peers may refuse the installation of a new chaincode version. To be independent of such peers, it is reasonable to spawn additional peers or to adapt the chaincode's endorsement policy.

Similar to the endorsement of ordinary chaincode transactions, transactions for instantiating or upgrading a deployed chaincode have to be endorsed by the participants of the channel [39]. Fabric utilizes separate and customizable policies for these transactions. That way, a consensus for the instantiation or upgrade of a chaincode itself needs to be reached, which in turn mitigates attacks by individuals, like those aimed at the compromisation of the chaincode implementation.

For curing vulnerabilities in our client logic, such as the CLI and the API, one has to establish an upgrade management process for these components, which is currently not

available. To ensure the compliance with standards and best practices, it is advisable to employ a well-known and mature packet management system such as Debian's *APT*. Then, updates of the client logic can easily be deployed to the subjects interacting with our system, consisting of proposers, approvers and targets. With that approach, it is even possible to fully automate the deployment process.

Lastly, vulnerabilities in Hyperledger Fabric itself have to be managed. Since Fabric serves as the foundation of our implementation, security flaws in the framework can quickly endanger crucial security requirements, such as tamper-resistance and traceability. The developers of Fabric describe a step-by-step procedure for upgrading the orderer and peer nodes to the latest versions available [50]. At the time of writing this thesis, there is no automated upgrade process for the Fabric components. Consequently, major parts of the process must be conducted manually, which is time-consuming and error-prone. However, the upgrade process can be supported by the Docker toolset, as the orderer and peer instances are typically managed with Docker.

We cannot make a general statement about the impact of particular vulnerabilities. In the worst case, a vulnerability can break essential security properties of our system, regardless of whether it originates from the implementation of chaincodes, the client logic or Fabric itself. Therefore, the different stakeholders participating in an instance of our system are responsible for keeping their peers, orderers and clients up to date, at best by enforcing an effective and efficient software upgrade process.

7.3.4 Auditing

Since Distributed Ledger Technology (DLT) inherently provides transparency and unforgeability with respect to the performed transactions, the participants are able to perform a convenient and reliable auditing process. Whenever there is the suspicion of a security-relevant incident or anomaly to have occurred, DLT provides means to investigate and analyze who has performed which transactions. Those incidents may result from the employment of weak policies, from a collusion of several participants in order to bypass security measures, or from errors in the chaincode and client logic itself.

With respect to Hyperledger Fabric, there are different ways to perform such an auditing process. The most elementary way is to use the official command-line tools, in particular `peer channel` to fetch blocks of a network's channel [47] and `configtxlator` to translate the blocks into a human-readable representation [41]. Working with these low-level tools can help to understand the internals of the ledger, but usually is inefficient for frequent and comprehensive audits.

Hyperledger Explorer[1] is a comprehensive suite for investigating the internals of a Fabric network. It is an official Hyperledger project and hosted by the Linux Foundation. After having been set up on a server or locally on a client machine, Explorer is accessible as a web service, consisting of a Graphical User Interface (GUI) and a Representational State Transfer (REST) API. The suite allows not only to conveniently inspect blocks and transactions of a network's channel, but also to retrieve detailed information on the nodes itself, such as their status and the chaincodes installed on them.

7.4 Limitations

As with every work, there are limitations to keep in mind when employing our system. The awareness and handling of these limitations is relevant for understanding and guaranteeing the security properties of our system.

7.4.1 Security Risks of Policies

First of all, we cannot effectively prevent weak or misused policies from mitigating certain security properties, since the design of particular policies highly depends on the environmental conditions and use cases where our system is employed. If our chaincodes are set up together with endorsement policies that violate certain security measures, for instance by requiring endorsements only from peers of the first organization and ignoring the second, the tamper-resistance provided by the Fabric network will be considerably weakened. The same holds for the policies on the application level, ACPs and VPs: if the number of approvals required for a set of targets, denoted by m, is set too low, it will be easier to activate malicious or flawed configurations. Similarly, attackers may exploit that some policies or policy rules are less strict than others and try to force an evaluation against the weaker one. Our simplistic approach with respect to the policy language and the policy evaluator contributes to obviate these attacks.

7.4.2 Collusion of Participants

Another limitation originates from the nature of distributed consensus systems: a collusion of a sufficiently large amount of participants can dominate the consensus process

[1] `https://www.hyperledger.org/projects/explorer/`

and therefore break security properties like tamper-resistance and Multi-Party Authorization (MPA). If multiple entities secretly agree upon the manipulation of a chaincode's execution result or upon the approval of a malicious configuration, there will be a reasonable chance that they are successful. Stronger policies and a larger amount of entities operating and controlling the consensus process can mitigate this kind of threat, but cannot eliminate it. Since the network provides transparency with respect to all conducted transactions and operations, however, it is possible to establish a frequent and automated auditing process in order to detect anomalies in hindsight. Based on this evidence, measures can be taken to address organizational and structural problems.

7.4.3 Compromisation of Targets

After multiple parties have authorized a configuration, it is automatically applied to the intended targets by the *handler* daemon that is shipped with our implementation. To guarantee the security properties of our system, it is crucial to ensure that only authorized configurations are applied. To prevent the activation of unauthorized configurations, the targets have to be locked, for instance by disabling remote shell access and by preventing any kind of physical access. As a consequence, nobody must be able to access the targets with administrative privileges directly. However, a target remains a critical point of failure: if attackers are able to effectively compromise it, no matter whether they operate from inside or outside, they will be able to manipulate its configuration at will. Currently, our solution cannot mitigate the compromisation of targets. Future approaches may address this issue on other levels, for instance by operating the same service redundantly on multiple targets, providing some kind of automated fallback in the case of a misconfiguration.

7.4.4 Consistency, Availability and Partition Tolerance

For every distributed storage system, one has to consider limitations with respect to the consistency, availability and partition tolerance it provides. The CAP theorem, which was proposed by Brewer in 1998 and formally proved by Gilbert and Lynch in 2002 [26], states that only two of these three properties can be fulfilled for a distributed storage system simultaneously. Consistency denotes that all nodes are able to receive the same data at the same point of time, whereas availability implies that every request has to result in a non-error response. Lastly, partition tolerance connotes that the system continues to operate even if the communication between arbitrary nodes is disturbed or broken. Since a peer-to-peer network as employed by Fabric has to cope with network outage, it must provide partition tolerance. According to the CAP theorem, there will

be two choices left if partitioning occurs: sacrificing availability and thus obtaining a CP system, or sacrificing consistency and thus obtaining an AP system. In practice, however, this is not a decision that is made a priori. Instead, the system can have either CP or AP characteristics depending on the kind of partition.

If the partition is of such a kind that a transaction cannot be fully endorsed according to the endorsement policy and/or no consent between the orderer nodes can be established, there will be an error and the ledger will not be updated at all, leading to no inconsistencies and thus CP. Hence, a deployment dependent on consistency has to establish endorsement policies and ordering mechanisms that are in line with its real-world requirements, preventing a ledger update if parties relevant for a certain transaction are cut off. In the worst case, however, there may be a partition where a transaction can be fully endorsed and a consent between the orderer nodes can be established, leading to a ledger update on the nodes of that partition only, which provokes inconsistencies and thus AP. But after the partitioning has been resolved, the nodes of the other partitions are able catch up on the updated ledger state by performing synchronization mechanisms, for instance via Fabric's built-in *gossip data dissemination protocol* [43].

7.5 Trade-Offs

With respect to the performance and security of our system, trade-offs have to be made. In general, it is not possible to achieve a maximum of performance and security at the same time, as both properties behave dichotomously to each other. A particular setup of our system may strengthen the security it provides while simultaneously decreasing its performance. In contrast, there are setups that focus on a higher performance, but cannot guarantee an equally high level of security. There is no silver bullet for this problem: the requirements in terms of performance and security have to be evaluated and implemented for each use case and IT environment separately.

The first trade-off has to be made regarding the ACPs and VPs in use. Generally speaking, stricter policies introduce a higher latency for the overall authorization process: with a strict ACP, the number of administrators entitled to propose a new configuration is limited to a minimum. On the one hand, this may increase security, as less administrators are able to propose faulty or malicious configurations. On the other hand, latency may increase, since the proposal of a configuration depends on a small set of actors. The same holds for a strict VP: requiring many participants to review and approve a configuration, including a variety of tests to perform, strengthens the

principle of redundant supervision and weakens the power of an individual, but leads to an increased processing time due to the high number of parties involved.

Another trade-off arises regarding the structure and setup of the Fabric network itself: only a sufficiently large number of endorsing peers and orderer nodes can provide tamper-resistance with respect to the execution of the configuration management process and the storage of data. If only few nodes are employed, the barrier for collusions and compromisations will be significantly lower. The specific amount of network nodes again depends on the use case, ranging from less than ten for small environments to several dozens for larger ones. However, a larger network comes at a price: reaching a consensus between a high number of nodes typically raises the computation and communication overhead, leading to negative effects on the overall performance. Besides, the employment of additional nodes may lead to an increase in operational expenses.

7.6 Future Work

On the basis of our system's design and implementation, there is future work to be done. In this section, we focus on further concepts for interpreting the semantics of configurations and for handling emergency cases.

7.6.1 Semantic Interpretation of Configurations

Currently, our system is not able to interpret the semantics of configurations. Although there is support for different configuration types, we are not able to reliably and precisely determine which software is configured in which respect by a proposed configuration. However, this semantic information could be highly useful for differentiating the authorization process: an *iptables* configuration unblocking a certain network port, for instance, may require the authorization of a larger number of parties than a configuration of the *Apache HTTP Server* adapting the log level.

The declarative and powerful configuration languages of well-known configuration management frameworks such as Ansible can provide a solid basis for a semantic analysis: depending on the configuration modules and directives used, it is possible to assess the targeted software and the configuration's impact. Our policy model consisting of ACPs and VPs can then be extended to take into account these attributes.

7.6.2 Emergency Cases

Another aspect not covered by our approach is the handling of emergency cases. According to our definition, an emergency case requires the immediate activation of a configuration change on a set of targets, without the enforcement of a time-consuming authorization process. Typical examples include the repair of a broken configuration that has led to an outage or vulnerability of vital parts of the IT landscape, in particular at times when no other administrators are available to review and approve that change, such as at night-time.

Each organization must decide on their own if they want to allow the execution of such an emergency procedure, since this feature involves the danger of being misused: the possibility to circumvent the principle of Multi-Party Authorization (MPA) in an emergency can lead to an even worse emergency if an attacker or amateur is at work.

There are two fundamental approaches to implement an emergency procedure. The first one is purely software-based: the *propose* operation of our system's management process would be enriched with an *emergency* flag. If this flag was set by a proposer, the corresponding CR would become valid immediately. Due to the transparency provided by the Distributed Ledger Technology (DLT), every operation performed as part of an emergency procedure would still be recorded in an unerasable and unforgeable way. Therefore, the necessity and consequences of an emergency procedure could be audited in hindsight. As an additional measure, our system could send a notification of the ongoing incident to an external system responsible for Security Information and Event Management (SIEM). Nevertheless, it is advisable to restrict the capabilities of an emergency procedure as much as possible. On the one hand, this may involve time-based restrictions, for instance by allowing emergency procedures only at that times of the day where very few administrators are available. On the other hand, only predefined configurations may be selectable for emergencies, like one that shuts down critical devices of the IT landscape to prevent vulnerabilities from being exploited.

The other approach involves physical actions and, as a last resort, is always feasible: in cases of emergency, the corresponding devices are accessed physically by an administrator. Physical access barriers such as a locking system may have to be passed to access critical devices. In order to disclose and document the physical access, some form of physical sealing can be introduced. After an administrator has opened a sealed envelope containing a key for the physical emergency access, for instance, evidence is created for third parties and auditors that an emergency procedure must have taken place.

Chapter 8

Conclusion

In conclusion, our research contributes a novel way of configuration management for networked devices. We designed and implemented **TANCS**, a **T**amper-Resistant and **A**uditable **N**etwork **C**onfiguration **S**ystem. By basing our system on the principle of Multi-Party Authorization (MPA), we require multiple administrators to review and approve a configuration change before it can be applied to the targeted devices. Therefore, no individual administrator is able to decide on a critical configuration on its own, mitigating insider attacks as well as accidental failures. To provide tamper-resistance with respect to the management process and the configuration storage, we leverage Distributed Ledger Technology (DLT). In particular, we employ Hyperledger Fabric, a DLT framework providing a distributed smart contract engine and a distributed ledger. With that approach, even the abuse of an individual's administrative privileges cannot negatively affect the operability of our system.

For a real-world deployment of TANCS, several security-relevant aspects have to be considered. On the one hand, the requirements for a configuration change to become valid need to be adapted to the target environment, since our system cannot protect against weak policies. Highly critical IT environments, like those in health care institutions or airplanes, are typically in need of stricter conditions for the validity of a configuration change than ordinary environments. On the other hand, the setup and structure of the Fabric network itself needs to reflect the real world's trust model: if several participants, who only share a limited amount of trust between each other, jointly manage the configurations for their device infrastructure, each participant will have to contribute own nodes to the Fabric network in order to understand, execute and audit the configuration management operations. Besides, the number of nodes in the Fabric network has to be sufficiently large to reduce the risk of compromisation and collusion.

Compared to the related approaches presented in chapter 4, TANCS combines a unique set of security properties into one system, as illustrated in the final overview in table 8.1. TANCS does not only offer MPA and auditability for configuration changes, as several others do, but also guarantees tamper-resistance due to the distributed process execution and data storage, while simultaneously providing a logically centralized configuration repository. Moreover, it allows to conduct federated configuration management across the boundaries of different stakeholders, which is especially useful for IT landscapes managed by multiple independent parties with limited trust, but joint responsibilities.

	TANCS	**Ansible**	**ACHEL**	**Transparency Overlays**
Authorization	✓	✓	✓	
Multi-Party Authorization	✓	(✓)	✓	
Auditability	✓	(✓)	✓	✓
Tamper-Resistance	✓		(✓)	✓
Distributed process execution and data storage	✓		✓	
Logically centralized configuration repository	✓			
Federated configuration management across stakeholders' boundaries	✓			
Semantic interpretation of configurations			✓	
Resistance to compromisation of targets				

TABLE 8.1: Final overview of the system's capabilities, based on the findings presented in table 4.1

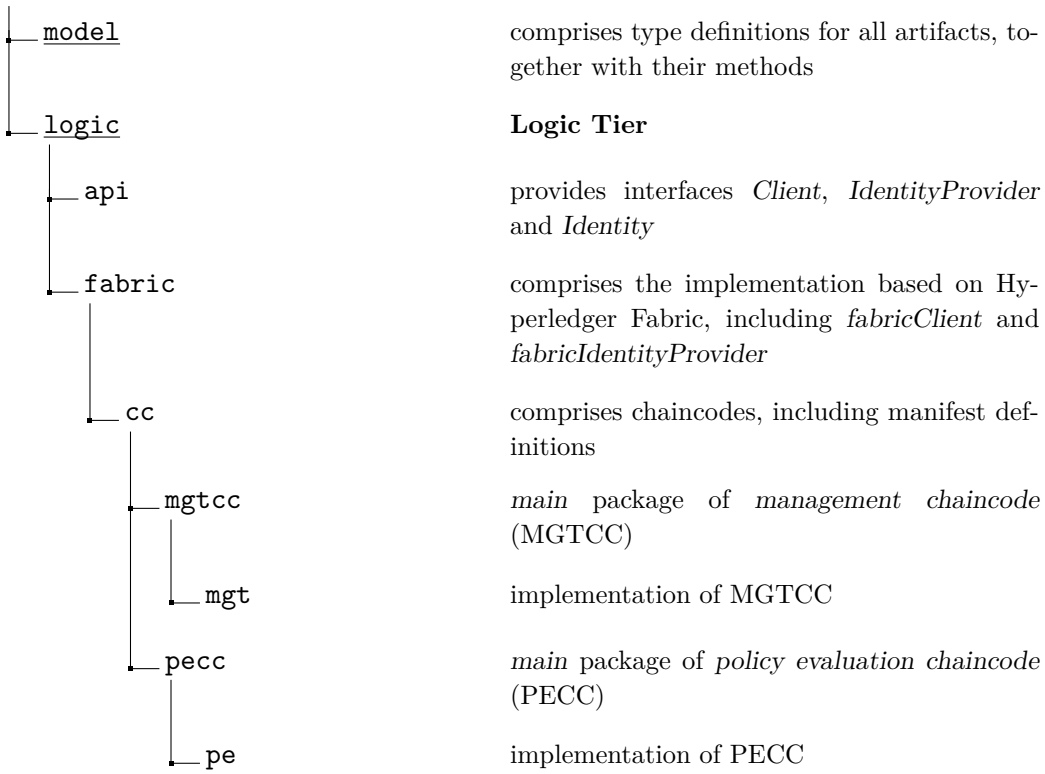
Our long-term vision is to expand the principle of MPA to several other areas. A highly relevant use case involves the certification process prevalent in Public Key Infrastructures (PKIs), which currently is conducted in a centralized manner and without the approval of multiple independent parties. By establishing a process where multiple Certificate Authorities (CAs) have to sign a requester's public key before it becomes regarded as valid by operation systems and web browsers, malicious behaviour regarding the issuance of certificates, as encountered regularly in the past, can be effectively restrained. In an even broader sense, we envision to design a generic and reusable service architecture, offering the principle of MPA in a tamper-resistant manner and with a focus on the efficient and convenient deployment to a multitude of different IT environments, such as the Internet of Things (IoT).

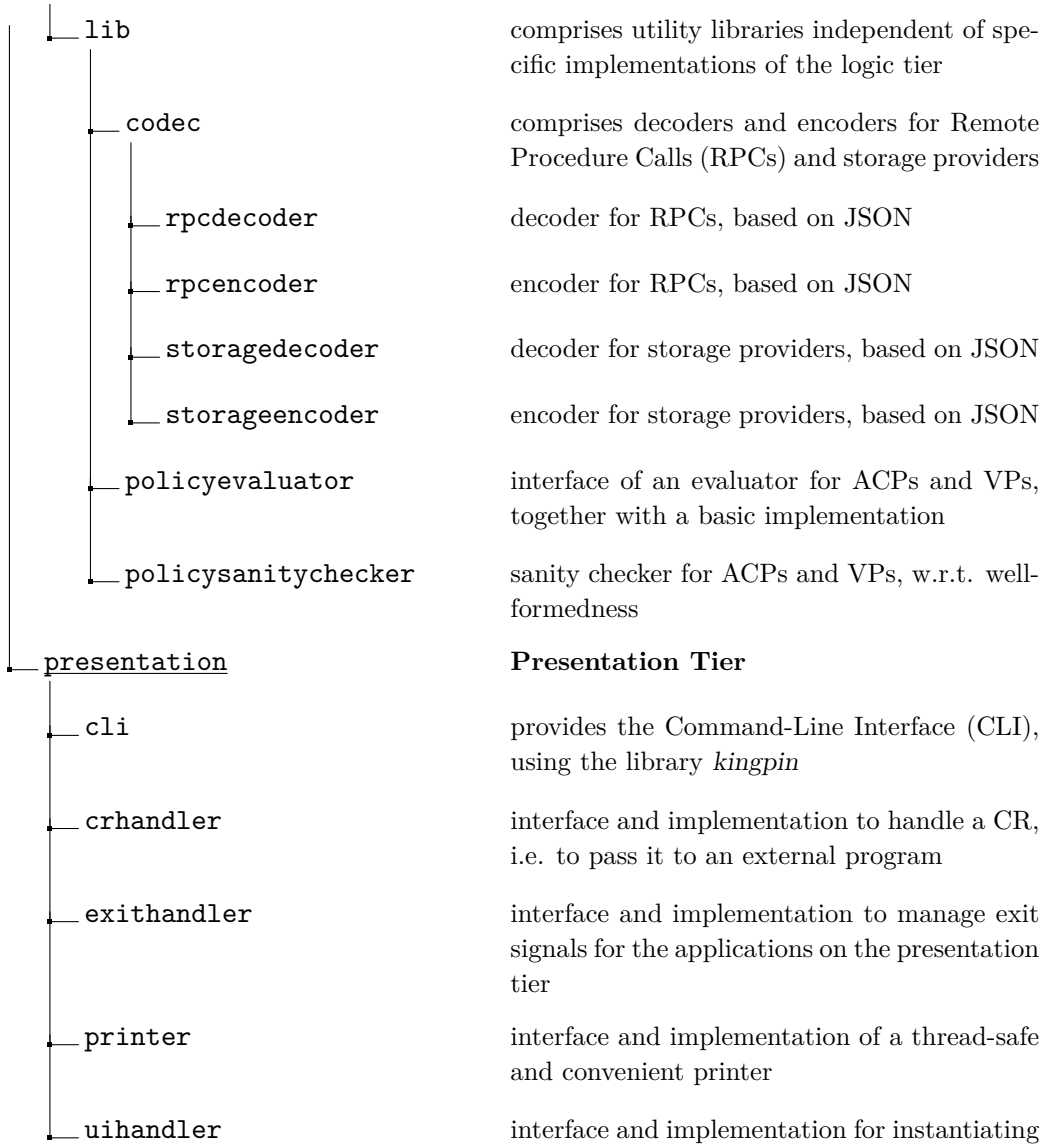
Chapter A

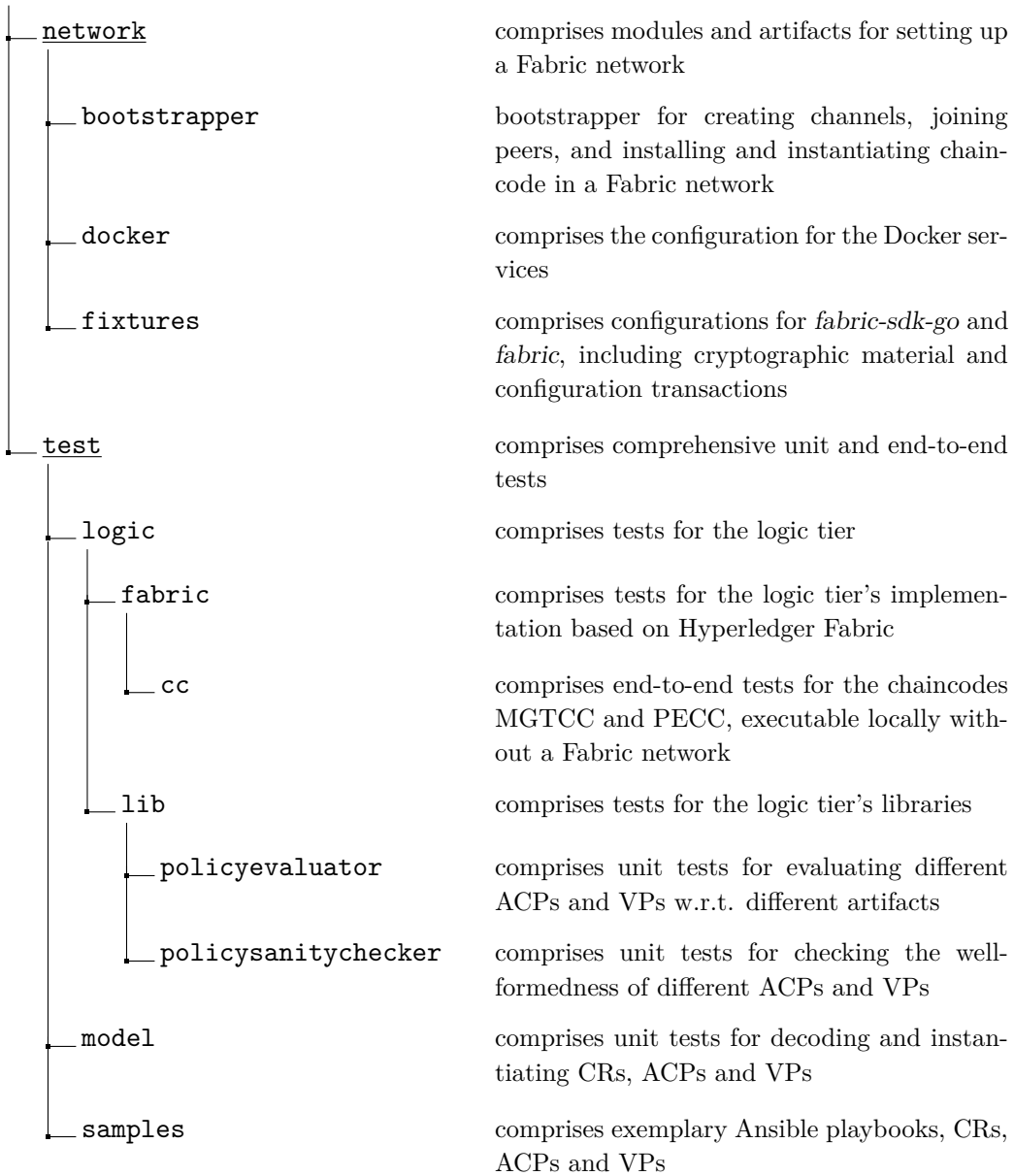
Appendix

A.1 Implementation

A.1.1 Structure of Go Packages




lib
comprises utility libraries independent of specific implementations of the logic tier
codec
comprises decoders and encoders for Remote Procedure Calls (RPCs) and storage providers
rpcdecoder
decoder for RPCs, based on JSON
rpcencoder
encoder for RPCs, based on JSON
storagedecoder
decoder for storage providers, based on JSON
storageencoder
encoder for storage providers, based on JSON
policyevaluator
interface of an evaluator for ACPs and VPs, together with a basic implementation
policysanitychecker
sanity checker for ACPs and VPs, w.r.t. well-formedness
presentation
Presentation Tier
cli
provides the Command-Line Interface (CLI), using the library kingpin
crhandler
interface and implementation to handle a CR, i.e. to pass it to an external program
exithandler
interface and implementation to manage exit signals for the applications on the presentation tier
printer
interface and implementation of a thread-safe and convenient printer
uihandler
interface and implementation for instantiating and initializing an identity provider, an identity and a client, serving as intermediary between the presentation and the logic tier


network
comprises modules and artifacts for setting up a Fabric network
bootstrapper
bootstrapper for creating channels, joining peers, and installing and instantiating chaincode in a Fabric network
docker
comprises the configuration for the Docker services
fixtures
comprises configurations for *fabric-sdk-go* and *fabric*, including cryptographic material and configuration transactions
test
comprises comprehensive unit and end-to-end tests
logic
comprises tests for the logic tier
fabric
comprises tests for the logic tier's implementation based on Hyperledger Fabric
cc
comprises end-to-end tests for the chaincodes MGTCC and PECC, executable locally without a Fabric network
lib
comprises tests for the logic tier's libraries
policyevaluator
comprises unit tests for evaluating different ACPs and VPs w.r.t. different artifacts
policysanitychecker
comprises unit tests for checking the well-formedness of different ACPs and VPs
model
comprises unit tests for decoding and instantiating CRs, ACPs and VPs
samples
comprises exemplary Ansible playbooks, CRs, ACPs and VPs

Bibliography

[1] Muneeb Ali et al. *Blockstack: A New Internet for Decentralized Applications*. 2017. URL: https://blockstack.org/whitepaper.pdf (visited on 05/09/2018).

[2] Elli Androulaki et al. "Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains". In: *EuroSys* (2018). URL: https://arxiv.org/pdf/1801.10228.pdf (visited on 04/28/2018).

[3] Ansible, Inc. *Ansible Documentation*. URL: https://docs.ansible.com/ (visited on 02/20/2018).

[4] Ansible, Inc. *ansible-playbook - Ansible Documentation*. URL: https://docs.ansible.com/ansible/latest/ansible-playbook.html (visited on 03/22/2018).

[5] Ansible, Inc. *Ansible Tower - Ansible Documentation*. URL: https://docs.ansible.com/ansible/latest/tower.html (visited on 03/03/2018).

[6] Ansible, Inc. *Intro to Playbooks*. URL: https://docs.ansible.com/ansible/latest/playbooks_intro.html (visited on 02/20/2018).

[7] Apache Software Foundation. *10.3.6. /db/_find - Apache CouchDB 2.1 Documentation*. URL: http://docs.couchdb.org/en/latest/api/database/find.html (visited on 03/23/2018).

[8] Apache Software Foundation. *5.1. Compaction - Apache CouchDB 2.1 Documentation*. URL: http://docs.couchdb.org/en/latest/maintenance/compaction.html (visited on 03/23/2018).

[9] Apache Software Foundation. *The Power of B-trees*. URL: https://guide.couchdb.org/draft/btree.html (visited on 05/12/2018).

[10] Sebastiano Battiato, Giampaolo Bella, and Salvatore Riccobene. "Should We Prove Security Policies Correct?" In: *Proceedings of the 1st International Workshop on Electronic Government and Commerce: Design, Modeling, Analysis and Security*. 2004, pp. 56–65. URL: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.224.8852&rep=rep1&type=pdf (visited on 03/21/2018).

[11] Alysson Bessani, João Sousa, and Eduardo E. P. Alchieri. "State Machine Replication for the Masses with BFT-SMaRt". In: *Proceedings of the 2014 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. DSN '14. Washington, DC, USA: IEEE Computer Society, 2014. URL: https://www.di.fc.ul.pt/~bessani/publications/dsn14-bftsmart.pdf (visited on 05/12/2018).

[12] Beth E. Binde, Russ McRee, and Terrence J. O'Connor. *Assessing Outbound Traffic to Uncover Advanced Persistent Threat*. 2011. URL: https://www.sans.edu/student-files/projects/JWP-Binde-McRee-OConnor.pdf (visited on 02/27/2018).

[13] Peter Bright. *Another fraudulent certificate raises the same old questions about certificate authorities*. 2011. URL: https://arstechnica.com/information-technology/2011/08/earlier-this-year-an-iranian/ (visited on 03/04/2018).

[14] Miguel Castro and Barbara Liskov. "Practical Byzantine Fault Tolerance and Proactive Recovery". In: *ACM Transactions on Computer Systems (TOCS)* 20.4 (Nov. 2002), pp. 398–461. URL: http://pages.cs.wisc.edu/~remzi/Classes/739/Papers/p398-castro.pdf (visited on 04/08/2018).

[15] Melissa Chase and Sarah Meiklejohn. *Transparency Overlays and Applications*. London, United Kingdom, 2016. URL: https://eprint.iacr.org/2016/915.pdf (visited on 03/03/2018).

[16] Chef Software, Inc. *chef-cookbooks/httpd: Library cookbook with Apache httpd primitives*. URL: https://github.com/chef-cookbooks/httpd/ (visited on 02/20/2018).

[17] Chef Software, Inc. *Chef Docs*. URL: https://docs.chef.io/ (visited on 02/20/2018).

[18] Chef Software, Inc. *chef-solo - Chef Docs*. URL: https://docs.chef.io/chef_solo.html (visited on 03/22/2018).

[19] Jeffrey Dean and Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. San Francisco, CA, USA, 2004. URL: https://research.google.com/archive/mapreduce-osdi04.pdf (visited on 03/23/2018).

[20] Thomas Delaet, Wouter Joosen, and Bart Vanbrabant. "A Survey of System Configuration Tools". In: *Proceedings of the 24th International Conference on Large Installation System Administration*. LISA'10. San Jose, CA: USENIX Association, 2010, pp. 1–8. URL: https://www.usenix.org/legacy/event/lisa10/tech/full_papers/Delaet.pdf (visited on 03/06/2018).

[21] Paul Dunphy and Fabien A.P. Petitcolas. “A First Look at Identity Management Schemes on the Blockchain”. In: *IEEE Security and Privacy Magazine* (2018). URL: https://arxiv.org/pdf/1801.03294.pdf (visited on 03/21/2018).

[22] European Directorate for the Quality of Medicines. *Change Control*. Strasbourg, France, 2017. URL: https://www.edqm.eu/sites/default/files/recommandation-omcl_16_122_r_change_control_web_publication.pdf (visited on 03/03/2018).

[23] David Ferraiolo, Serban Gavrila, and Wayne Jansen. *Policy Machine: Features, Architecture, and Specification*. 2014. URL: https://nvlpubs.nist.gov/nistpubs/ir/2014/NIST.IR.7987.pdf (visited on 04/09/2018).

[24] Colin Fletcher and Terrence Cosgrove. *Innovation Insight for Continuous Configuration Automation Tools*. 2015.

[25] Free Software Foundation, Inc. *GNU Coding Standards: Command-Line Interfaces*. 2016. URL: https://www.gnu.org/prep/standards/html_node/Command_002dLine-Interfaces.html (visited on 03/14/2018).

[26] Seth Gilbert and Nancy Lynch. “Brewer’s Conjecture and the Feasibility of Consistent, Available, Partition-Tolerant Web Services”. In: *SIGACT News* 33.2 (June 2002), pp. 51–59. ISSN: 0163-5700. URL: https://www.comp.nus.edu.sg/~gilbert/pubs/BrewersConjecture-SigAct.pdf (visited on 04/04/2018).

[27] Gideon Greenspan. *MultiChain Private Blockchain - White Paper*. 2015. URL: https://www.multichain.com/download/MultiChain-White-Paper.pdf (visited on 02/25/2018).

[28] Benjamin Hof and Georg Carle. *Software Distribution Transparency and Auditability*. 2017. URL: https://arxiv.org/pdf/1711.07278.pdf (visited on 03/03/2018).

[29] Jeffrey Hunker and Christian W. Probst. “Insiders and Insider Threats - An Overview of Definitions and Mitigation Techniques”. In: *Journal of Wireless Mobile Networks, Ubiquitous Computing, and Dependable Applications (JoWUA)* 2.1 (Mar. 2011), pp. 4–27. URL: http://isyou.info/jowua/papers/jowua-v2n1-1.pdf (visited on 03/06/2018).

[30] Intel Corporation. *PoET 1.0 Specification - Sawtooth v1.0.1 documentation*. 2017. URL: https://sawtooth.hyperledger.org/docs/core/releases/latest/architecture/poet.html (visited on 02/26/2018).

[31] Hussein Jebbaoui et al. “Semantics-based Approach for Detecting Flaws, Conflicts and Redundancies in XACML Policies”. In: *Comput. Electr. Eng.* 44.C (May 2015), pp. 91–103. ISSN: 0045-7906. DOI: 10.1016/j.compeleceng.2014.12.012.

URL: http://csm.beirut.lau.edu.lb/~rharaty/pdf/J4.pdf (visited on 03/21/2018).

[33] Daniel Kraft et al. *Namecoin Wiki*. URL: https://wiki.namecoin.org/ (visited on 05/09/2018).

[34] Ben Laurie, Adam Langley, and Emilia Kasper. *RFC 6962 - Certificate Transparency*. 2013. URL: https://tools.ietf.org/html/rfc6962/ (visited on 03/03/2018).

[35] Wenjie Lin. "Secure Multi-Party Authorization in Clouds". PhD thesis. The Ohio State University, 2015. URL: https://etd.ohiolink.edu/!etd.send_file?accession=osu1429041745 (visited on 03/06/2018).

[36] Linux Foundation. *Architecture Explained - hyperledger-fabricdocs master documentation. Ordering service nodes (Orderers)*. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/arch-deep-dive.html#ordering-service-nodes-orderers (visited on 03/25/2018).

[37] Linux Foundation. *Architecture Explained - hyperledger-fabricdocs master documentation. Validated ledger and PeerLedger checkpointing (pruning)*. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/arch-deep-dive.html#post-v1-validated-ledger-and-peerledger-checkpointing-pruning (visited on 03/23/2018).

[38] Linux Foundation. *Chaincode for Developers - hyperledger-fabricdocs master documentation*. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/chaincode4ade.html (visited on 03/25/2018).

[39] Linux Foundation. *Chaincode for Operators - hyperledger-fabricdocs master documentation. Instantiate*. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/chaincode4noah.html#instantiate (visited on 05/11/2018).

[40] Linux Foundation. *Channels - hyperledger-fabricdocs master documentation*. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/channels.html (visited on 03/25/2018).

[41] Linux Foundation. *configtxlator - hyperledger-fabricdocs master documentation*. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/commands/configtxlator.html (visited on 04/06/2018).

[42] Linux Foundation. *Endorsement policies - hyperledger-fabricdocs master documentation*. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/endorsement-policies.html (visited on 03/25/2018).

[43] Linux Foundation. *Gossip data dissemination protocol - hyperledger-fabricdocs master documentation*. URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/gossip.html (visited on 04/04/2018).

[44] Linux Foundation. *Introduction - Sawtooth v1.0.1 documentation.* URL: https://sawtooth.hyperledger.org/docs/core/releases/latest/introduction.html (visited on 02/26/2018).

[45] Linux Foundation. *Membership Service Providers (MSP) - hyperledger-fabricdocs master documentation.* URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/msp.html (visited on 03/25/2018).

[46] Linux Foundation. *MSP Identity Validity Rules - hyperledger-fabricdocs master documentation.* URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/msp-identity-validity-rules.html (visited on 03/25/2018).

[47] Linux Foundation. *peer channel - hyperledger-fabricdocs master documentation.* URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/commands/peerchannel.html (visited on 04/06/2018).

[48] Linux Foundation. *shim - GoDoc. type ChaincodeStubInterface.* URL: https://godoc.org/github.com/hyperledger/fabric/core/chaincode/shim#ChaincodeStubInterface (visited on 05/13/2018).

[49] Linux Foundation. *Transaction Flow - hyperledger-fabricdocs master documentation.* URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/txflow.html (visited on 03/25/2018).

[50] Linux Foundation. *Upgrading Your Network Components - hyperledger-fabricdocs master documentation.* URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/upgrading_your_network_tutorial.html (visited on 04/06/2018).

[51] Linux Foundation. *Welcome to Hyperledger Fabric - hyperledger-fabricdocs master documentation.* URL: https://hyperledger-fabric.readthedocs.io/en/release-1.1/ (visited on 03/25/2018).

[52] Kief Morris. *Infrastructure as Code: Managing Servers in the Cloud.* O'Reilly Media, 2016. ISBN: 9781491924358.

[53] Satoshi Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System.* 2008. URL: https://bitcoin.org/bitcoin.pdf (visited on 02/23/2018).

[54] Harish Natarajan, Solvej Karla Krause, and Helen Luskin Gradstein. *Distributed Ledger Technology (DLT) and Blockchain.* Washington, D.C., USA, 2017. URL: http://documents.worldbank.org/curated/en/177911513714062215/pdf/122140-WP-PUBLIC-Distributed-Ledger-Technology-and-Blockchain-Fintech-Notes.pdf (visited on 02/23/2018).

[55] Object Management Group. *Interface Definition Language, v4.1.* 2017. URL: https://www.omg.org/spec/IDL/4.1/PDF/ (visited on 03/16/2018).

[56] OTRS, Inc. *OTRS Comparison*. 2017. URL: https://www.otrs.com/wp-content/uploads/2017/11/019-EN_Comparison_OTRS_6.pdf (visited on 03/03/2018).

[57] Serguei Popov. *The Tangle*. 2017. URL: https://iota.org/IOTA_Whitepaper.pdf (visited on 02/23/2018).

[58] Puppet. *Language: Basics*. URL: https://puppet.com/docs/puppet/5.4/lang_summary.html (visited on 02/20/2018).

[59] Puppet. *Man Page: puppet apply - Puppet (PE and open source) 5.5 | Puppet*. URL: https://puppet.com/docs/puppet/latest/man/apply.html (visited on 03/22/2018).

[60] Puppet. *Puppet Documentation*. URL: https://puppet.com/docs/ (visited on 02/20/2018).

[61] Robert N. Rose. *The Future Of Insider Threats*. 2016. URL: https://www.forbes.com/sites/realspin/2016/08/30/the-future-of-insider-threats/ (visited on 03/06/2018).

[62] João Sousa, Alysson Bessani, and Marko Vukolić. *A Byzantine Fault-Tolerant Ordering Service for the Hyperledger Fabric Blockchain Platform*. Lisbon, Portugal, 2017. URL: https://arxiv.org/pdf/1709.06921.pdf (visited on 04/09/2018).

[63] Bart Vanbrabant, Thomas Delaet, and Wouter Joosen. "Federated Access Control and Workflow Enforcement in Systems Configuration". In: *Proceedings of the 23rd Conference on Large Installation System Administration*. Leuven, Belgium: DistriNet, Dept. of Computer Science, KU Leuven, 2009. URL: https://www.usenix.org/legacy/event/lisa09/tech/full_papers/vanbrabant.pdf (visited on 04/03/2018).

[64] Gavin Wood. *Ethereum Yellow Paper: a formal specification of Ethereum, a programmable blockchain*. 2014. URL: https://ethereum.github.io/yellowpaper/paper.pdf (visited on 02/25/2018).

FIN.[1]

[1] For now.