TÉCNICO LISBOA

Distributed Ledger Technology to Enable Secure Management of IT Infrastructures

Miguel Rodrigo Moreira Oliveira

Thesis to obtain the Master of Science Degree in

Information Systems and Computer Engineering

Supervisor: Prof. Rui António dos Santos Cruz

Examination Committee

Chairperson: Prof. Nuno João Neves Mamede
Supervisor: Prof. Rui António dos Santos Cruz
Member of the Committee: Prof. José Carlos Martins Delgado

November 2021

This work was created using LaTeX typesetting language
in the Overleaf environment (www.overleaf.com).

Acknowledgments

I would like to thanks my parents for their support during this academic journey that culminated with the writing of this Thesis. For all the encouragement, love, friendship throughout the years, both in the good and specially in the bad moments, enabling me to do my best and to get the best education possible.

To my brother, that despite not fully understanding what ”i do”, is always present, being one of the most important people in my life, with a bond that only brothers may understand.

To my friends, my second family, that have been always present, either in person or virtually, not only during the development and writing of this Thesis, but also by taking part in my life, both in the good and bad moments, encouraging, supporting, and helping me grow as a person.

I would also like to thank Professor Rui Santos Cruz for his insight, guidance and knowledge sharing during the development of this Thesis, ultimately making this work possible, and enabling me to gain extra knowledge in this field.

To each and every one of you – Thank you.

Abstract

IT Infrastructures have grown in both size and complexity. To help administrators to manage their infrastructure, several Infrastructure Management Tools have been created. However, none of them implements a secure a traceable log of changes that can bring accountability to the management of such infrastructures. On the other hand, recent research and development in blockchain technologies have allowed for the creation of Distributed Ledgers that can, in theory, solve the problem by providing a secure, immutable and traceable ledger that can store the changes that the infrastructure management tools apply to the infrastructure. In this Thesis, we develop a proof-of-concept solution that incorporates a Distributed Ledger, Hyperledger Fabric, and infrastructure management tools, Ansible and Terraform, to prove the suitability of the usage of a distributed ledger to provide a secure inventory and log of changes in a manner that enables for traceability and accountability for all modifications to the infrastructure, while also providing user identity management and verification of business constraints.

Keywords

Infrastructure Management; Blockchain; Distributed Ledger Technology; Software Defined Infrastructure; Provisioning; Ansible; Terraform; Hyperledger Fabric; Traceability; Accountability;

Resumo

As Infraestructuras de TI têm crescido tanto em tamanho como complexidade. Para ajudar os administradores a gerir a sua infraestructura, diversas Ferramentas para Gestão de Infraestructuras foram criadas. No entanto, nenhuma delas implementa, de uma maneira segura, um registo de alterações que consiga providenciar rastreabilidade na gestão de tais infraestructuras. Por outro lado, investigação e desenvolvimento recentes na área das tecnologias de *blockchain* permitiram a criação de Notários Distribuídos que conseguem, em teoria, resolver o problema, providenciando um notário seguro, imutável e rastreável, que consegue guardar as alterações que as ferramentas de gestão de infraestructuras aplicam nas infraestructuras. Nesta Tese, desenvolvemos uma solução prova-de-conceito que incorpora um Notário Distribuído, *Hyperledger Fabric*, e ferramentas de gestão de infraestructuras, *Ansible* e *Terraform*, para provar a possível utilização de um notário distribuido para providenciar um inventário e registo de alterações seguros, de uma maneira que permita a rastrabilidade e apuramento de responsabilides para todas as modificações à infraestructura, e, ao mesmo tempo, providenciando ferramentas para a gestão de identidades de utilizadores e verificação de regras de negócio.

Palavras Chave

Gestão de Infrastructuras; Blockchain; Tecnologia de Notários Distribuídos; Infraestructura Definida por Programa; Provisionamento; Ansible; Terraform; Hyperledger Fabric; Rastreabilidade; Responsabilidade;

Contents

1 Introduction **1**

1.1 Motivation . . . 5

1.2 Goals . . . 6

1.3 Organization of the Document . . . 7

2 Background **8**

2.1 Similar Tools and Solutions . . . 9

2.2 Distributed Ledger Technology . . . 9

2.2.1 Information Storage . . . 10

2.2.2 Blockchain . . . 11

2.2.2.A Public and Private Blockchains . . . 11

2.2.3 Smart Contracts . . . 11

2.2.4 Hyperledger . . . 12

2.2.4.A Hyperledger Fabric . . . 12

2.2.5 Using Distributed Ledger Technology . . . 13

2.3 Infrastructure Management . . . 15

2.3.1 Direct Connection . . . 15

2.3.2 Infrastructure Management Tools . . . 16

2.3.2.A Specialized Tools . . . 17

3 Proposed Solution **18**

3.1 Application Requirements . . . 19

3.2 Approach . . . 20

3.2.1 General Architecture . . . 21

3.2.1.A Broker Module . . . 22

3.2.1.B Ledger Module . . . 22

3.2.1.C Tool Module . . . 22

3.2.2 General Data Flow . . . 22

3.2.2.A Login . . . 22

3.2.2.B Read/Write to the Ledger . . . 23
3.2.2.C Tool execution . . . 23
3.3 Design Decisions . . . 25
3.3.1 Infrastructure Management Tools . . . 25
3.3.1.A Provisioning . . . 25
3.3.1.B Configuration Management . . . 26
3.3.2 Distributed Ledger . . . 26
3.3.3 Programming Language . . . 27
4 Implementation **28**
4.1 Development Methodology . . . 29
4.2 Environment . . . 29
4.3 Hyperledger Fabric . . . 30
4.3.1 Certificate Authorities . . . 31
4.3.2 Identity Generation . . . 31
4.3.3 Peer Configuration . . . 32
4.3.4 Orderer Configuration . . . 33
4.3.5 Channel Creation . . . 33
4.3.5.A Genesis Block . . . 33
4.3.6 Chaincode Development . . . 34
4.3.6.A Asset Type Management . . . 37
4.3.6.B Asset Management . . . 37
4.3.6.C Dependency Management . . . 38
4.3.6.D Applied Tool Management . . . 38
4.3.7 Chaincode Installation . . . 39
4.4 Modules Development . . . 40
4.4.1 Module Architecture . . . 41
4.4.2 Ledger Module . . . 42
4.4.3 Broker Module . . . 43
4.4.4 Tools Modules . . . 45
5 Evaluation and Result Analysis **47**
5.1 Test Scenarios . . . 49
5.2 Scenario 1 - Authentication . . . 50
5.3 Scenario 2 - Normal Workflow . . . 51
5.3.1 Asset Management . . . 51
5.3.2 Tool Management . . . 53

5.3.2.A Execution Request . . . 54
5.4 Scenario 3 - Authorization . . . 57
5.5 Scenario 4 - Dependency processing . . . 58
5.5.1 Automatic Dependency Detection Shortfall . . . 59
5.6 Scenario 5 - Rollback of tool applied actions . . . 60
5.6.1 Tool Rollback Limitations . . . 60
5.6.2 Proven Solution . . . 61
5.6.3 Integration with our solution . . . 62
5.7 Load and throughput . . . 62
5.8 Result Discussion . . . 66
6 Conclusion **68**
6.1 Objectives . . . 69
6.2 Conclusions . . . 70
6.3 System Limitations and Future Work . . . 71
Bibliography **71**
A Sample Smart Contract Code **75**

List of Figures

1.1 Logical Location of the tool 6

2.1 General RBAC model architecture 14

3.1 General Solution Architecture 21
3.2 Login Data Flow 23
3.3 Read/Write Requests Data Flow 23
3.4 Tool Invocation Data Flow 24

4.1 Host-VM-Container architecture 30
4.2 Module Architecture and Data Flow 41

5.1 User and Admin Permissions Diagram 57
5.2 Response times for different request types 64
5.3 Throughput for different request types 65

List of Tables

2.1 Comparison between provisioning and configuration management tools 16

4.1 Broker Module public API 44

Listings

4.1 Registration and Enrollment of an Identity 32
4.2 Asset Structure 35
4.3 Asset Structure 36
4.4 Installation of a Smart Contract 39
4.5 Fabric SDK Connection 42
5.1 Successful Login Request 50
5.2 Unsuccessful Login Request 50
5.3 Successful Asset Registration 52
5.4 Successful Asset Deletion 53
5.5 Successful Asset Registration 54
5.6 Refused Request due to lack of permissions 58
5.7 Refused Asset removal due to Dependency check 59
A.1 Asset Registration 75
A.2 Remove Asset 77
A.3 Get Asset 78

Acronyms

API Application Programming Interface

CA Certificate Authority

CD Continuous Delivery

CI Continuous Integration

CLI Command-Line Interface

DAG Directed Acyclic Graph

DL Distributed Ledger

DLT Distributed Ledger Technology

GUI Graphical User Interface

HTTPS Hypertext Transfer Protocol Secure

IaaS Infrastructure as a Service

ID Identifier

JSON JavaScript Object Notation

MSP Membership Service Provider

OS Operating System

RAM Random Access Memory

RBAC Role Based Access Control

SDK Software Development Kit

SDN Software Defined Networking

TLS Transport Layer Security

TPS Transactions per Second

VM Virtual Machine

1

Introduction

Contents

1.1 Motivation 5

1.2 Goals 6

1.3 Organization of the Document 7

Currently, due to the continuous development of new and more complex software systems, tools, and applications, employing new development methods like Continuous Integration (CI) [1] and Continuous Delivery (CD) [2], the IT Infrastructures have grown in size and complexity, both in large datacenters, typically of service providers, and in smaller ones of private businesses that still maintain their own physical infrastructures. New technologies were introduced to ensure the necessary adaptability and versatility of the Infrastructures to enable them to be constantly changing on a logical level with minimal physical level changes, aiming to separate the underlying hardware from the software running in those infrastructures [3].

From the computing perspective, the most promising and used technologies are based on virtualization. **Virtualization** is "the creation of a virtual (rather than actual) version of something, such as an operating system, a server, a storage device or network resources. It allows a single physical resource (such as a server, operating system, application, or storage device) to function as multiple logical resources" [4]. This means it is possible to have the same physical server running more that one operating system. This brings several benefits in cost-of-ownership, enables a much faster and easier creation of new (logical) systems, enables portability of the virtualized systems, sometimes being possible to move a virtual resource between physical machines without turning off that resource, ultimately making management easier and more effective.

Other popular technology is **containerization**. This technology is very similar to virtualization but, instead of virtualizing entire Operating Systems (OSs), enables the virtualization of specific applications or services, avoiding the problem of having the overhead of an entire OS running just to execute said service. With this technology, it is possible to have a system, with only one OS, executing several services while isolating each of them, ensuring the security and separation between services [5] [6] [7].

From the networking perspective new technologies were also introduced with the same goal of decoupling the physical **data** plane from the control plane, introducing the term Software Defined Networking (SDN). This separation enables all the network to be managed from a central location, instead of static configurations in each networking equipment via well defined protocols like OpenFlow [8], which makes possible to create dynamic networks that automatically adapt to the needs (i.e., ensuring connectivity between certain hosts) and to possible failures, routing packets through alternative routes, minimizing downtime. The SDN concept can be also extended to manage the virtual networks created when more and more Virtual Machines (VMs) are being executed in a single physical host, by using virtual switches that can be managed using flows. This enables the network to be constantly re-configuring itself to ensure the needed connectivity without the need for physical interaction and ensuring fast adaptation to new requirements. [9] [10]

By combining the benefits of software defined computing infrastructure, using virtualization and containerization technologies, and the SDN concept, was possible to create large infrastructures that can

be controlled and managed from one point (with the necessary redundancies) and that enable users to have a limited access to a part of the computing and storage resources of the infrastructure. This also enables the users to create and destroy resources (i.e. Virtual Machines) on demand and with total abstraction of the underlying hardware and management software via Application Programming Interfaces (APIs). This concept of total abstraction is known today as the cloud, and it can be implemented in very large and public scales, like Google Cloud [11], AWS [12] and more, or in a private environment, usually created within a company for self use [10].

To manage these software stacks, several tools were created. These tools usually accept as input plans or scrips, written in a declarative language that may or not be specific for the tool (instead of being specific for the hardware and software running in the infrastructure). The tools are capable of converting these plans into actions that are executed against the several components of the infrastructure, and then can detect all errors and misconfigurations, allowing for the easy monitoring of the changes being applied and enable for the easy detection of inconsistencies across the several assets in the infrastructure [3].

However, since different tools have different objectives, it is common practice for the administrators of the infrastructures to to use different tools of their preference to completely cover the lifecycle of the resources. This can lead to inconsistencies (for example, trying to configure a resource that has not been provisioned), lack of awareness about the global state of the infrastructure, and also lack of a trusted environment where auditability and traceability are ensured. Furthermore, since little or no history of changes is kept, it is very difficult to trace back individual actions, consequently being very difficult to revert them or investigate who executed some change and when [3].

Some of these problems can be solved by using an orchestrator, able to monitor and control the activity of each tool and to keep a resource inventory of the Logical and Physical Infrastructure status, with information about the available resources, currently allocated resources and their ownership, and even some knowledge base related with those resources. This information is typically stored in a database, where it can be readily modified and accessed, allowing for the system administrators to keep track of the individual assets of the infrastructure, improving therefore the awareness on their status, and, at the same time, providing information about ownership and utilization of the resources [3].

Although, at first, the idea of a database to keep a resource inventory can appear to solve all problems described, it does not. A database can be very good to provide information on the current state of the infrastructure. However, this type of solution does not ensure the traceability of the all the actions that brought the infrastructure to the current state (i.e., the actual state stored in the database) and does not ensure an immutable historic record of previous actions over the infrastructure or over the database (for example, it is possible for someone to delete information from the inventory, allowing for the cover up of malicious activities without leaving a trace).

From a security and traceability perspective, as the infrastructures grow larger and more valuable

services are run on them, the need for ensuring the security of the infrastructure increases. At the same time, as more people need to integrate the management teams, it is increasingly important to ensure everyone can know what has been done, what is pending and how to revert changes. This can be solved by introducing access control to the management tools and by keeping a log of all the changes that have been committed [3].

It is also important to ensure that the generated logs are immutable to prevent malicious users from deleting or changing the logs. The use of Distributed Ledger Technology (DLT) and a Distributed Ledger (DL) can help to ensure these security constraints while also taking an active role in the verification of the state of the infrastructure and verification of business logic, for example, dependency verification when modifying assets. This behaviour is ensured by using chaincode, or smart contracts, that are in fact general purpose code that the ledger executes in order to register or modify new entries in the state database, while also building the logs of actions.

1.1 Motivation

This Thesis' project focus is in the middle layer of software present in such environments, between the hardware and the users. More specifically, it focuses on orchestrating different tools while ensuring that all the changes are registered and verified by the ledger.

Due to different design philosophies and goals, there are different tools that aim to provide the same of different subsets of the provisioning, configuration and deployment processes. Since different tools satisfy different objectives, it is normal to use several such tools in a single environment. Due to the need of using different tools to accomplish different tasks, it is then needed for the engineers in charge of managing the infrastructure to use different tools independently, making the management harder, more difficult to keep track of all the changes and ultimately making easier to commit mistakes and more difficult to spot and correct them.

Another issue with this lack of centralized control is the lack of traceability of changes that are made using each individual tool, and by whom. Most of the tools, even though they can analyze and sometimes store the state of the infrastructure, they do not keep track of who triggered the changes, when those changes occurred, and the history of changes. With the ever growing size of the infrastructures and teams, it is essential to keep a timeline of all changes to the infrastructure and who is responsible for each.

With the traceability perspective in mind, it is important to have then a sequence of the actions that were made, and to make sure that that sequence is immutable and tamper proof, to eliminate the risk of a malicious agent making changes to the infrastructure and then wipe the records of those changes. A DL suits this requirement perfectly, by allowing decentralized recording (improving safety and redundancy)

and by providing an immutable sequence of blocks by design.

1.2 Goals

Our objective with this work is to evaluate the suitability of a Blockchain-based DL to provide a secure log storage and inventory (state database) of the infrastructure, while also harnessing its chaincode capabilities to ensure business logic verification automatically, such as ensuring that new actions do not break dependencies between different infrastructure assets.

To make this evaluation a Proof-of-Concept tool that acts as infrastructure status tracker will be developed, employing a DL to maintain a log of changes as well as a secure inventory that represents the state of the infrastructure. In order to track changes being applied, it will also act as an orchestrator between the different infrastructure management tools, allowing the user to have a single point of contact to manage the entire infrastructure.

The DL will also be programmed with proof-of-concept chaincode to evaluate all the actions taken over the infrastructure, ensuring their correctness, verification of permissions and dependency tracking between assets.

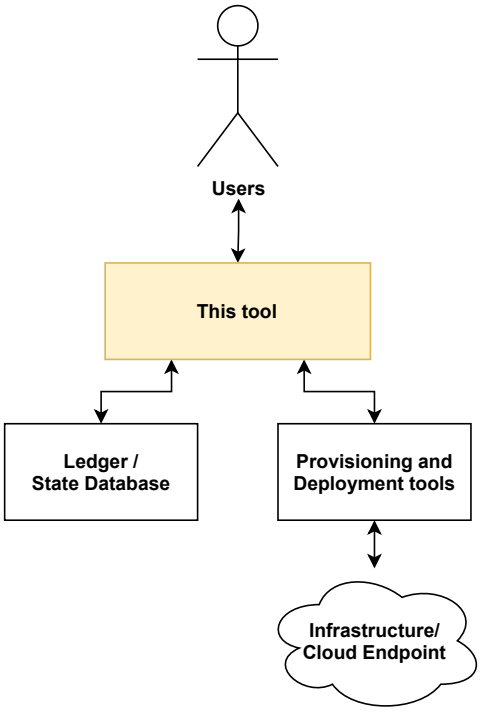


Figure 1.1: Logical Location of the tool

The tool is to be used as a orchestrator for the various provisioning and management tools, while also acting as an authentication point and ledger. As Figure 1.1 illustrates, the tool will serve as an

abstraction layer between the User and the underlying tools. The tool is to be developed following a modular architecture, ensuring that it is possible to add support to new southbound tools without needing for a major reprogramming. The tool will be developed to support connections to Ansible, Terraform as a proof-of-concept, but ensuring the possibility of connecting to different tools with just the addiction of an API.

The tool will receive as input configuration files written in the language the Southbound tools may use, forwarding them to the corresponding tool, while also analyzing its output in order to register the taken actions in the ledger. It will be also possible for the user to manually insert new actions or assets into the state database, provided that the modifications comply with the chaincode.

From the security perspective, the tool is to be able to connect to some Ledger, via a universal API. Hyperledger Fabric [13] will be used in this proof of concept, as it is a DL that can provide both the traceability and authentication needed. The tool will then register in the Ledger all of the actions taken over the infrastructure, forming a flow of actions that can be retrieved later and presented to the user.

1.3 Organization of the Document

This thesis is organized as follows:

- Chapter 1 - Introduction: describes the motivation, the problem, and the goals of the thesis;
- Chapter 2 - Background: explores the current state of the art, and related works;
- Chapter 3 - Proposed Solution: Describes the chosen architecture and the requirements for implementation;
- Chapter 4 - Implementation: Describes the implementation of the tool, together with the explanation for several design choices;
- Chapter 5 - Evaluation and Result Analysis: proposed methodology for the evaluation of the work and its evaluation;
- Chapter 6 - Conclusion: presents the conclusions taken from the developed and presented work.

2

Background

Contents

2.1 Similar Tools and Solutions 9

2.2 Distributed Ledger Technology 9

2.3 Infrastructure Management 15

To develop this tool, it is important to understand the current state of the art in the related areas. Firstly, we will discuss the existence of similar tools and **systems that enable a centralized management and logging point for an infrastructure**. Secondly, as it is the main area of analysis of this project, it is needed to understand the DLT, the DLs and, more specifically, the **workings and capabilities of the DL** that will be used in this proof of concept, while also comparing it to some other relevant DLs. We will also present some examples from the literature of the usage of DLs, in areas different from IT Infrastructure Management, but at the same time with similarities to the work we present on this project. Thirdly, we will **evaluate, compare and discuss different Infrastructure Management Tools**, since they ensure the connection between this tool and the physical infrastructure, and will be present in this proof-of-concept tool as an example of the adaptability of this tool.

2.1 Similar Tools and Solutions

There are several similar tools and toolsets that aim to centralize an infrastructure management and provide a central logging database with authentication and access control capabilities. However, we could not find any that was either open sourced or free, since most of them are commercial solutions with high costs and developed by companies that use them as a source of profit. Although by being closed sourced and payware the available information is sparse and not technical, avoiding a meaningful comparison with the proposed project, this shows that there is a need for tools with these objectives in the industry. Additionally, from our research, none of those tools harness the capabilities of DLTs to improve their functionality or security, instead relying on traditional technologies such as normal database systems. It must also be noted that many of these tools are developed and published by the hardware vendors (for example Cisco [14], Juniper [15], HPE [16] and Dell [17]), tailor made to their hardware and environments, while some other frameworks are hardware agnostic, supporting multiple vendors and even integrating with existing open sourced tools like Nuage Virtualized Services Platform [18] or Ansible Tower [19]. In general, these frameworks, similarly to the one proposed in this project, work as a middle layer between the users and the tools that manage the hardware and software, sometimes open sourced, like the ones presented below in section 2.3.2, or have those tools already integrated in the framework and connect to the hardware via custom and proprietary protocols (e.g., custom implementations of Command-Line Interfaces (CLIs) and/or APIs).

2.2 Distributed Ledger Technology

DLT, as its name implies, is a **technology that enables a ledger to be distributed over several machines while maintaining synchronization between all of them** [13, 20]. A DL has several advan-

tages over a traditional centralized ledger, both in redundancy of data storage and security, due to the distributed design philosophy and the secure design of Blockchain [21].

From the security and traceability standpoint, it is needed to ensure the logging of information in a highly available, append-only database. DLT ensures this premises by using physically distributed storage and computing devices, even in an untrustworthy environment. There are many implementations of this technology, with different objectives and employment of different designs, based on Blockchains or Directed Acyclic Graphs (DAGs). In general, DLs are highly available, append-only distributed databases that work on untrustworthy environments, where Byzantine failures can happen, like crashed or unreachable nodes, occurrence of big network delays, and even malicious behavior of nodes can happen. DLs are comprised of separate Nodes that work together to maintain a consistent state of the ledger across all Nodes, that is replicated in every Node [22, 23]. There are many comparative studies about DLs like [22] and [24]. We will focus this work on the Hyperledger Fabric [13] solution because of its large acceptance, performance, modularity, available documentation and general architecture and assurances that are closely related with the ones of this project (e.g., high Transactions per Second (TPS) rate, high security and restricted participation) [23].

2.2.1 Information Storage

Distributed ledgers usually maintain a database where assets are stored. Depending on the specific implementation of the DL, these assets can represent different entities, such as a cryptocurrency, a physical asset or even files or abstract concepts. The DL implementation will determine what is possible to store in that database, the format of the database (e.g., relational database, key-value store) and how the users interact with said data. This database is usually called the State Database [13]. All the changes to the state database are registered in what is in fact the ledger, and that is the main focus in the DLT. The two most used technologies by distributed ledgers to store the changes are DAGs and Blockchains. Both these data structures are comprised of nodes, and relations between nodes. The main difference between them is the number of child nodes of each node. In DAGs, each node can have any number of child nodes whereas in a Blockchain each node only has one node, forming a chain [3, 13].

The motivation behind these nodes is that each node represents a modification to the state of the Ledger [13]. For this application it is only conceivable to have one state at a time, since it is in fact an inventory and from an Infrastructure perspective having more than one concurrent inventory does not make sense, we will further investigate the usage of Blockchain based Distributed Ledgers.

2.2.2 Blockchain

A **Blockchain is a distributed and decentralized data structure that works as chain of data nodes where each node ensures the integrity of the previous one using cryptographic functions** [21,25].

This data structure is comprised of several nodes where each one stores user set data, but also information about the node itself and the preceding node in the chain. By storing an Hash [26] of the preceding block in the chain, we can ensure the the preceding block in the chain has not been modified since it was referenced in the current block since it would change the hash of the preceding block and consequently the hash of the current block, because, since the hash of the preceding block is part of the current block, any change in that hash would also change the hash of the current block. This way, each block ensures the integrity of all the previous blocks, and any change in a random block would break the whole chain. This turns a blockchain in an append only database [21, 25].

Apart from the specific details about the blockchain definition, it is also important to understand the two main types of blockchains in existence: **public** and **private**.

2.2.2.A Public and Private Blockchains

A **Public Blockchain** is a blockchain implementation where all the information comprised in the blockchain, and the blockchain itself are public [13,25]. Also, the computers used to manage and store the chain are not trusted. By being public, this enables anyone to modify the chain, usually by appending new blocks. This is the type of blockchain used in most cryptocurrencies, where everyone can read and write to the blockchain, registering new transactions. Usually these blockchains use some algorithm to regulate the addition of new blocks to the chain such as proof-of-work, as used on Bitcoin. These algorithms rely on verifiable parameters such as the passage of time, computing and processing work and usage of storage space [13].

In contrast, a **Private, or Permissioned, Blockchain** is an implementation where all the information that comprises the blockchain is private and all access to the chain, being it for reading or writing, is controlled. In order to ensure the access control and privacy, all the computing nodes related to the blockchain must be identified and secured, and all users must also be identified. This is usually done using cryptographic certificates and keys that identify and ensure the identity of people and machines [13]. Since all access is controlled, in this type of Blockchains normally there is no algorithm to regulate the addiction of new blocks to the chain, making it faster and less resource demanding.

2.2.3 Smart Contracts

Smart contacts, also called Chaincode, are programs that are executed in order to change the state of the ledger [20]. While in a traditional database it is usual to simply commit information to the database,

in a DL all the changes to the state database are committed via smart contracts. These smart contracts are coded in some programming language, being it general purpose or not, depending on the specific implementation of the DL [20, 23]. This software layer enables custom processing of all the actions submitted to the ledger, enabling both fine grained access control (while the DL itself usually only does more general access control) for example such as a **Role Based Access Control**, or **Attribute Based Access Control**, and the verification of business logic, for example verification of constraints for the deletion or modification of assets based on the state of the ledger or even external factors [20, 27, 28]. It is important to note that all transactions, both write or read, must be implemented as functions in a smart contract, and, to be executed, must be executed by calling the corresponding function on the smart contract [13].

2.2.4 Hyperledger

There exist many Blockchain based DLs that, although made to fulfill similar requirements, have in fact different implementations and design choices, mainly about the programming language in which they are written, that can affect the supported languages for the smart contracts and execution speed, modular or monolithic design, that can improve adaptability to different setups, and the consensus algorithms implemented to manage and replicate the blockchain over all the computing nodes.

In this project we will analyze in more detail the Hyperledger Family, that is a family of several DL implementations, each with specific features to specific implementation scenarios and different levels of modularity and adaptability, that often increase in exchange for more complex configuration and setups. The family is composed of 6 different DL implementations. The Hyperledger website[1] provides a good summary of each of them. The two implementations that would make sense to use in this solution would be both Hyperledger Fabric and Hyperledger Sawtooth, due to their modularity and high flexibility, that enables complex Smart Contracts, written in general purpose programming languages, their high performance, and adaptability of their state database. However, Hyperledger Fabric will be analyzed in more detail in section 2.2.4.A, and the reasons to choose it for this solution will be presented later in Chapter 3.

2.2.4.A Hyperledger Fabric

Hyperledger Fabric is a “modular and extensible open-source system for deploying and operating permissioned blockchains” [13]. Since it is intended to be used as a part of bigger solutions and systems, it is very configurable, modular, and offers complete APIs in Golang, Java, Javascript and Typescript that enable it to be controlled from other systems. From an architecture perspective, Fabric has two types of nodes, **Peers and Orderer**s.

[1] https://www.hyperledger.org/use/distributed-ledgers accessed on 6th September 2021

Peers are responsible for the execution of the smart contracts, as well as maintaining the state database together with a copy of the Blockchain. For each installed Smart Contract, each Peer creates a docker container where all executions of said smart contract will take place. The documentation calls for at least two Peers in any deployment for redundancy reasons [13].

The Orderer nodes are responsible for the ordering of the blocks and actions in the chain. The Orderer nodes also verify that the executions of the smart contracts terminated with success and are also responsible for some of the permissions management. The documentation recommends a minimum of three Orderer nodes in any deployment due to the need of consensus between Orderers in order to commit changes to the chain [13].

Another important component of any Fabric deployment is the Identity Management. Each user and machine connection to the deployment has to have an identity, in a form of a cryptographic key pair, together with a Certificate signed by a trusted Certificate Authority (CA). These files are then organized in an Membership Service Provider (MSP), that is a folder structure that identifies some identity (user or machine). The Hyperledger Fabric project provides an implementation of a CA that is customized to automatically produce the cryptographic material in the format required by Fabric (MSP). However, any CA can be used, provided that the operator manually organizes the cryptographic material to be compliant with the format required. All identities in Fabric are organized in organizations, that can be implemented to reflect different real life organizations/companies, or simply to provide some separation and organization in the logical Fabric Network [13].

As explained in detail in [13], the main difference in Hyperledger Fabric in comparison to other DLs is the workflow for any given transaction. Instead of the traditional *Order* - *Execute* flow, Hyperledger Fabric uses a *Execute* - *Order* - *Validate* flow that increases throughput and reduces execution times for the transactions. It also helps mitigating some issues with smart contracts such as non-termination and infinite looping.

2.2.5 Using Distributed Ledger Technology

AS explained in [20], since DLs can provide a trusted environment in untrusted and opaque environments, it can be used as a base for the processing of inter-organizational business processes. These processes can be fully represented in smart contracts, since they are programmed with general purpose programming languages that can execute logic and conditions. Since the translation from the business process's logic to the smart contracts can be modeled, the inverse translation is also possible, enabling the monitoring of the processes by people that do not need to understand the logic behind a DL, by presenting them with a Graphical User Interface (GUI). Since the management of IT Infrastructures can be also represented as business models, a similar process can be employed to enable monitoring of IT infrastructures using a DL and smart contracts.

It is also possible to use Hyperledger Fabric and its smart to contracts to enable user authentication and Attribute-Based access control. As the authors in [29] demonstrate, this technology can ensure the required auditability for Access Control Systems. Using a blockchain as base technology, this system also ensures a high level of transparency. The study also provides an experimental performance evaluation that shows this system can process large numbers of requests. This indicates that it should be possible to use Hyperledger Fabric's technology in this project in order to enable user authorization and authentication with acceptable performance and fulfilling the goals.

Similarly, the work in [30] presents an investigation for Access Control for Knowledge Management Systems using Blockchain. In this paper, a Role Based Access Control (RBAC) model, whose general architecture is shown on Figure 2.1, is developed, using the Elliptic Curve Digital Signature Algorithm as the base for key generation and authentication of the users, as it is widely accepted. The innovation comes from the use of a blockchain and its smart contracts to manage both the authorization and roles, and the management of the knowledge. Since the blockchain is **tamper proof, secure and distributed**, this advantages directly increase the security of the whole model. The paper also describes each of the smart contracts and steps in the process in a detailed manner. This can be of great use in this project, by providing a secure method for both user authentication and authorization for the execution of actions over the infrastructure and logging of those actions (knowledge in the model).

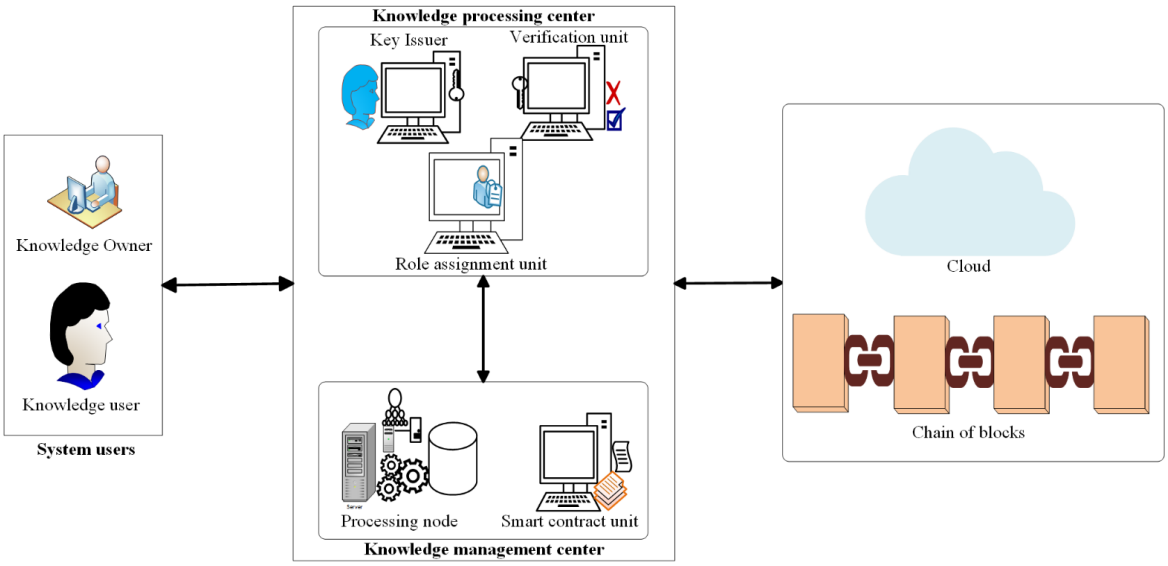


Figure 2.1: General RBAC model architecture

As seen in [31], the authors present the problem of (Distributed) Databases' tampering detection, and propose some methods and technologies to provide tampering detection, including the use of One-Way Cryptographic Hash Functions, Digital Watermarking, Audit Logs, Page Carving, and the use of Decen-

tralized Databases. Although these methods are already well known, the paper discusses Blockchain and Blockchain based Ledgers as solution to complement those existing methods, since this technology can provide an immutable ledger with high degrees of security and dependability by design. In our paper, we also propose a blockchain based solution as a complement for a database system, although aiming for traceability assurance, but relying on the same principles of immutability and dependability.

The work presented in [27] also focus on the benefits of integrating blockchain a based solution into already existing systems. In this paper, the Hyperledger Fabric, one of the major Blockchain based Distributed Ledger Projects, is presented as a tool to enable supply chain management. The possibility of running code integrated with the blockchain, the so called chaincode or smart contract, enables the verification and execution of business logic and supply chain specific conditions in order to automate most of the tasks related to the management of the products. The introduction of smart contracts to manage assets is also considered in our proposed solution, since it enables automatic verification of inventory conditions to accept or abort some new inventory changing operations.

The authors in [28] demonstrate the advantages of using a Distributed Ledger, based on blockchain, as a foundation for a platform for Pharmaceutical Cold Chain Management. In that paper, the authors provide an example of how the smart contract technology can be used to verify real world conditions. The proposed platform uses the blockchain as a ledger for the tracking of products and also smart contracts that verify the packaging conditions of the products. By reading information from sensors close to the package, the platform can automatically flag the package and abort the tracking process

2.3 Infrastructure Management

Since our goal is to develop a tool that can centralize and monitor tasks and actions taken over the infrastructure, it is important to analyze how to do so.

2.3.1 Direct Connection

One possibility would be to manually develop an ssh and API client that would connect to each and every system on the infrastructure. This would be a bad approach and against current trends in infrastructure management, since this approach has several drawbacks, such as:

- Since different systems have different APIs and different ssh commands, it would be needed for the user to still know all different commands and endpoints of the different systems;
- Due to the differences in the systems, it would be impossible to execute similar actions in different systems with deep refactoring of the scripts and instructions;

On account of the lack of desired uniformity, the industry has veered towards the development of tools that create an abstraction layer over the management of the infrastructure.

2.3.2 Infrastructure Management Tools

The infrastructure management tools aim to create, via an abstraction layer, a common format in which the infrastructure operator can specify the changes to be made in a standardized language, that the tool will then translate in the specific commands and API calls exposed by the systems.

Table 2.1: Comparison between provisioning and configuration management tools

Tool	Objective	Language	Client Agent	Contributors	Stars
Ansible	Configuration Management	Procedural (YAML)	No	5177[a]	46.1k
Chef	Configuration Management	Procedural (Ruby)	Yes	638[b]	6.4k
Puppet	Configuration Management	Declarative (PuppetDSL)	Yes	548[c]	6k
Heat	Provisioning	Declarative	No	384[d]	367
Terraform	Provisioning	Declarative (HCL)	No	1531[e]	24.9k

[a] https://github.com/ansible/ansible accessed on 21st December 2020
[b] https://github.com/chef/chef accessed on 21st December 2020
[c] https://github.com/puppetlabs/puppet accessed on 21st December 2020
[d] https://github.com/openstack/heat accessed on 21st December 2020
[e] https://github.com/hashicorp/terraform accessed on 21st December 2020

Must be noted that there are two different types of tools. Provisioning tools have as objective the provisioning of new resources, being them virtual machines, containers, networks, etc. Configuration Management tools are developed to help in the configuration of those resources, by executing actions in the virtual resources themselves (e.g., installing software, deploying configurations). This further confirms the need of more than one tool to fully manage the complete infrastructure (e.g., using Terraform to provision the resources and Ansible to configure them).

There are also differences between the tools in regards to human interaction. While all of them use configuration files as input, they can use different languages, and different logical approaches (Declarative or Procedural).

For the Configuration Management tools, it is important to note the necessity of a client agent for some of the tools. Since the tool cannot manage a resource that does not have the client agent installed, it is necessary to install that agent before the use of the tool. This must be done manually or with some other automatized method. As such, agent-less tools are easier to deploy, by only needing to be installed in the control host.

Since all these tools are open sourced, the number of contributors and stars on the repositories will reflect the the size of the community supporting the tool, which, in turn, will influence the documentation, information and guides available for said tool. It can also represent the tools that are more likely to incorporate new technologies and have faster development and bug fixes.

Table 2.1 summarizes main the characteristics of the most used and well known provisioning and configuration management tools. These tools are used both in private and public clouds to enable the automatic and scripted provisioning and configuration of machines by connecting via APIs to the underlying infrastructure management software. All tools presented are open source and/or free to use.

2.3.2.A Specialized Tools

It is, however, important to note that there are more tools for the management of infrastructures that are either specialized to more specific components of the infrastructure, such as to manage containers, SDN controllers, or proprietary tools to manage proprietary solutions, such as tools from hardware vendors to manage their own Servers. All of these are not further detailed here due to their smaller relevance in the general infrastructure and not being representative of the majority of the usage of infrastructure management tools. Additionally, since the objective of this thesis is to develop a proof-of-concept tool, it makes no sense to include such specialized and specific tools.

3

Proposed Solution

Contents

3.1 Application Requirements 19

3.2 Approach 20

3.3 Design Decisions 25

As it is briefly presented in Chapter 1, our aim is to develop a proof-of-concept tool to investigate the viability and benefits of utilizing a DL to maintain an inventory of an IT Infrastructure, and the possibility of this tool to be integrated with other infrastructure management tool to provide a central, but distributed and secure, point of management of an infrastructure while enabling accountability and traceability of changes made by operators.

3.1 Application Requirements

As said previously, the systems needs to address the security concerns naturally related to the management of the infrastructure primarily, but it is also important to ensure both speed and efficiency, and also adaptability to enable customization. A list of requirements was created to help the fulfilment of the goals:

- The system must be developed using a **microservices architecture**, deploying **modules** with simple objectives, improving adaptability.
- All modules should be **stateless**, in order to enable them to be more resilient to crashes and reboots;
- It is needed for the modules that handle communication to the infrastructure management tools to be **plug-and-play**, enabling the addition of support to new tools without the need to change code in other modules.
- The Ledger communication module must also be **plug-and-play**, to enable the utilization of different ledgers by creating just the ledger module responsible for the interface with the Ledger itself.
- All the modules' communication will be done via REST APIs, that must be **Tool/Ledger agnostic** in order to support the plug-and-play philosophy.
- The user communication with the tool will be made via a **REST API**.
- It must be possible for the user to **register the creation, modification or deletion of assets** in the infrastructure **manually**.
- The user will be enabled to submit plans for the infrastructure management tools via our tool, that will be responsible for the automatic registration on the ledger and implementation of the plans using the underlying infrastructure management tools.
- The tool must provide **user access control**, via user attributes (i.e., user attribute and admin attribute), that is supported and verified by the ledger. **The ledger must refuse any changes or queries made by non-authenticated or non-authorized users**.

- The ledger must be responsible, by using chaincode, of the **verification of the possibility of modification and deletion of assets based on dependencies** (e.g., if the asset B depends on asset A, asset A cannot be deleted before the dependency is removed or asset B is deleted).
- Any actions taken over the infrastructure using the connected tools **must not be implemented before they are registered in the ledger and approved** by it.
- The tool must be able to **register in the ledger and show to the user the return status of the infrastructure management tools** even if they fail.
- If the execution of some action fails, the tool must update the ledger so that the **inventory database continues to represent the real status of the infrastructure**.
- If it is detected that some infrastructure management tool has failed to implement an action, our tool must be able to **retry that action** a predefined number of times to try to **circumvent temporary failures**, granted that the infrastructure management tool grants **idempotency of applied actions** (e.g., if some plan is applied more than one time, the output must be the same as applying it just once).

3.2 Approach

Having in mind the requirements defined in section 3.1 the goal is to to develop a solution that harnesses the security and data storage benefits of a DL, by using it to store the inventory database for the infrastructure and make the necessary verifications to approve or deny infrastructure changes and inventory read operations (write/read operations) based on constraints expressed in smart contracts that will verify both user permissions and business constraints.

Since the presented solution is a proof-of-concept, the aim will be to implement and present key features that will be proven or disproven as viable, and, if viable, will indicate the viability of more complex features that are the evolution and more tight specifications of the ones implemented in this proof-of-concept. The main features to explore, according to the requirements will then be:

- **Attribute based access control** - In our proof of concept only two roles will be implemented (user and admin) but this will show the possibility to implement a much larger array of attributes, and with attribute hierarchy (i.e., admin is a ”member” of the users);
- **Dependency creation and checking** - In this prototype, only direct hardware dependencies will be implemented (i.e., a VM will depend on its Host). However this will prove the possibility of implementation of a more advanced dependency detection algorithm, more tightly related to the tools used and the infrastructure;

- Only one DL will be supported and implemented. However, since the APIs will be generic, any **similar DL can be possibly used**;

- Similarly, **only two infrastructure management tools will be incorporated**: **Ansible** and **Terraform**, but, as the Tools API will also be generic, **any tool can be incorporated**, even different types of tools, such as SDN controllers;

3.2.1 General Architecture

In order to satisfy the requirements for adaptability and modularity, a general architecture for the solution was designed. As shown on Figure 3.1, there are three main types of modules: **Broker**, **Ledger** and **Tool**. Each of these modules has a specific function, and the communication between them is, as specified in the diagram, done via REST APIs.

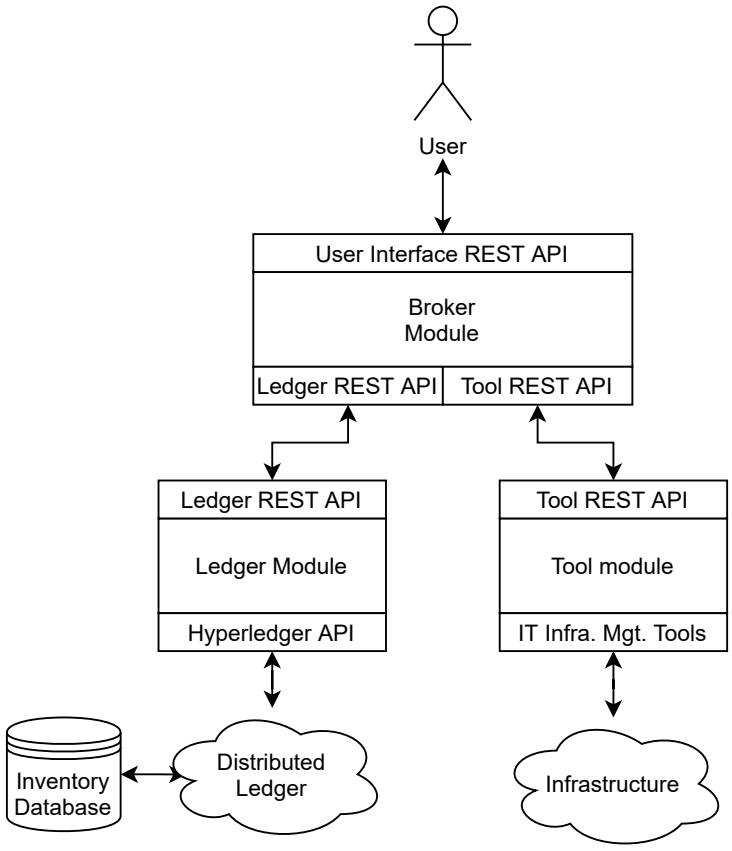


Figure 3.1: General Solution Architecture

We will now discuss in more detail the function of each of the modules.

3.2.1.A Broker Module

The **Broker module** acts as a central **routing** module for information. It handles the user interface, and relays the requests and information to the corresponding modules. The **processing done in this module is to be kept at a minimum** to keep it as generalized as possible, to enable it to accept connections to different tools and ledgers.

3.2.1.B Ledger Module

The **ledger module** is responsible for the **implementation of the DL API**, and to **convert all requests and information sent by the Broker module to the specific Ledger requests**. The processing done here should also be kept as a minimum, as only related to the conversion between request types. It is important to note that **the logic implemented in chaincode is part of the Ledger itself, and it is not present in this module**.

3.2.1.C Tool Module

The **Tool module** is the one that will make all processing that is **tool specific**. This module will **receive requests from the broker** module using the common tool API, **process those requests** and **execute the tools to fulfill them**. The module will then also **parse the tool's response** and convert it back to the common types present in the Tool's API, to be sent back to the Broker.

It is important to be noted that there may be **more than one tool modules** in the same deployment of this tool, as each tool module implementation only connects to one tool, and the **implementation of the module is tool dependant**.

3.2.2 General Data Flow

To ensure the modularity of this solution, specification of the interactions between the different modules is necessary. There are three main types of interactions between the user and the tool: **Login**, **Tool Execution**, **Read/Write information from the Ledger**. Although these interactions and data flows will be further explored and specified in Chapter 4, here we present an overview of said interactions.

3.2.2.A Login

Since the tool will support **Session based** access, a Login functionality is needed. As depicted in Figure 3.2, to login itself, the user will send its credentials to the broker module, that will send them to the ledger module, where they will be sent to the Ledger, running a specific login function on a Smart Contract that will make the necessary verifications to authenticate the user and provide a Session

Identifier (ID). That Session ID is then returned to both the Broker Module and the user for subsequent connections.

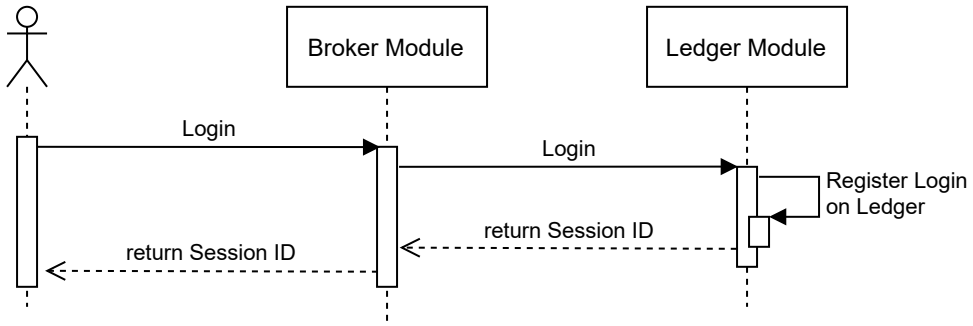


Figure 3.2: Login Data Flow

3.2.2.B Read/Write to the Ledger

One of the Infrastructure Management actions that the user can make is to **directly read or write to the ledger**. This can happen if the user wants to check the status of some asset (read) or register manual interactions that were not made with a Infrastructure Management Tool connected to our tool, or manually made (i.e., installing a new physical server). In this case, the user has to already have a Session ID that will accompany all its requests in order to authenticate and authorize itself. As presented in Figure 3.3, the user will again make the request to the broker, that will redirect the request to the Ledger module that will run the corresponding Smart Contract functions in order to fulfill the request. The Smart Contract Function's return value will then be redirected back, through the broker module, to the user.

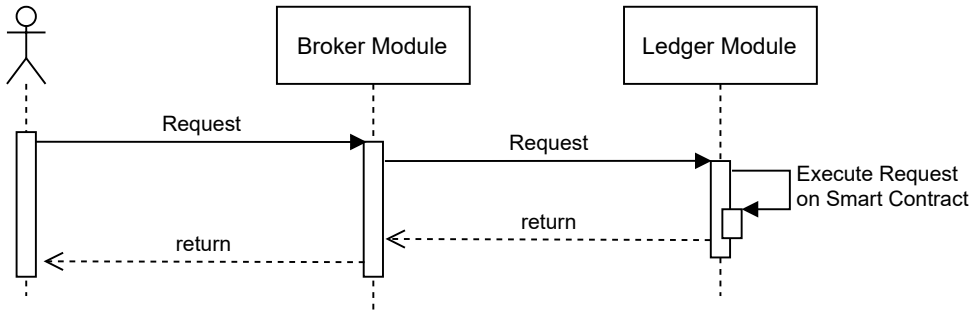


Figure 3.3: Read/Write Requests Data Flow

3.2.2.C Tool execution

The most important feature of our tool is to **automate the invocation of Infrastructure Management tools**, while keeping all the actions taken **registered** and **verified** by the ledger. Also in this case, the

user already has to have a Session ID that will accompany all its requests in order to authenticate and authorize itself. As shown on Figure 3.4, the user will make a request to invoke the tool, this request will include the necessary plan files to the tool execution. The request will then be verified for a valid session ID by the broker and sent to the Tool module to execute a dry-run, where the tool will verify the validity of the plans and return an estimation of the changes that will be made. Those changes are then sent to the ledger, by the broker module, that will verify if they are possible and valid using the business logic present in the smart contracts. It will also register the actions as being planned but not executed. The results of both the dry-run of the tool and the ledger return will then be sent to the user for final confirmation. If the user confirms the intent to execute said tasks/plans, the broker will then command the tool module to execute the Infrastructure management tool, this time committing changes to the infrastructure. The return values from the tool will then be processed by the broker and sent to the ledger for registration in the Ledger and Inventory. In the end, the user will get a summary of the changes.

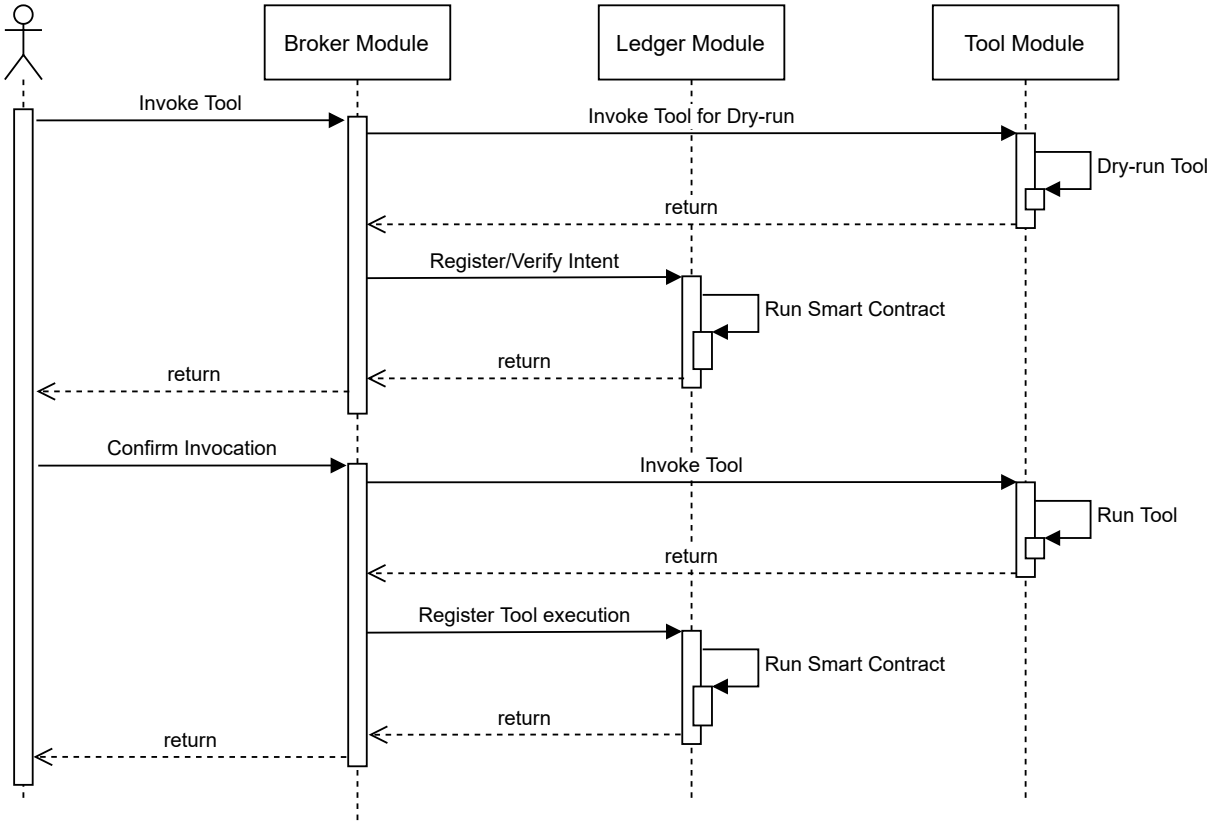


Figure 3.4: Tool Invocation Data Flow

Even tough the presented tool will be a proof-of-concept, design choices were taken into consideration to make it reflect the most recent tendencies both in software development and tool usage. In the next section we will present the taken considerations in the various decisions.

3.3 Design Decisions

Due to the immense variety of solutions in existence, it was needed to make choices about three main aspects in this solution: **Programming Language**, **DL implementation** and integrated **infrastructure management tools**. In the next sections we will present the reasoning for the choices taken for each aspect.

3.3.1 Infrastructure Management Tools

As presented on Section 2.3.2, there are many solutions and tools that aim to streamline and uniformize the process of **management of the infrastructure**. For this proof-of-concept tool, we chose to focus on the **general purpose tools of provisioning and configuration management** since they represent the management of most part of the infrastructure. This choice leaves behind container-specific tools, SDN controllers and other more specific management tools since they are less used in a general computing infrastructure and can always be implemented as new modules since our **proposed tool will be generalized to be able to process requests from any type of tool**.

Since there exists several general purpose tools, we choose to select one tool for each branch of operations: Provisioning and Configuration Management.

3.3.1.A Provisioning

As an example of a **provisioning** tool, we chose **Terraform**[1] due to its high popularity and ease of use. Due to its **high popularity and usage in the industry**, development for this tool is fast, with **thousands of contributors** to its public repository, while being maintained by HashiCorp[2], a well known and established company in the infrastructure business.

From a technical perspective, Terraform, due to its high popularity, already implements APIs to connect and **provision resources in the major providers**, such as OpenStack[3], Google Cloud[4], AWS[5] and Microsoft Azure[6], enabling users to **provision resources in any of those Infrastructure as a Service (IaaS) providers** with a common configuration file.

[1] https://github.com/hashicorp/terraform, accessed on 21st December 2020
[2] https://www.hashicorp.com/, accessed on 10th September 2021
[3] https://www.openstack.org/, accessed on 10th September 2021
[4] https://cloud.google.com/, accessed on 10th September 2021
[5] https://aws.amazon.com/, accessed on 10th September 2021
[6] https://azure.microsoft.com/, accessed on 10th September 2021

3.3.1.B Configuration Management

As an example of a **configuration management** tool, we choose **Ansible**[7] also due to its high popularity, that, along the support from RedHat[8], maintainer of the project, that also provides a paid version, is one of the most used configuration management tools. Once again, due to the very high popularity and contributor count, Ansible implements, as independent modules, abstraction layers to enable the **management of a very large count of OSs**, together with their OS **specific commands**, such as package managers, and **physical equipment**, such as network devices. This large array of supported devices is a strong indicator of the acceptance of this tool in the industry, since its main objective is to provide an abstraction layer to enable the management of devices with a common format, and that abstraction layer must be implemented manually for each new supported endpoint (device, OS).

3.3.2 Distributed Ledger

The Ledger to be used will be **Hyperledger Fabric**. This solution is an implementation of a Blockchain-based DL, part of the Hyperledger family. Firstly, we chose to use a member of the Hyperledger[9] family due to the high popularity, compared to other Permissioned Blockchain based DL, and the support from the Linux Foundation[10], a major player in the universe of open source solutions. This makes the members of this family active developed and documented solutions, an important aspect due to the current research in the blockchain solutions area and in the security area, both closely related to these solutions.

From the Hyperledger family, we chose to use Hyperledger Fabric in our solution due to several factors:

- It has the biggest community of users and contributors, which contribute to several factors:
  - The comprehensive and in-depth documentation;
  - Fast implementation of the newest security patches and optimizations;
  - Fast implementation of new features;
  - Fast correction of bugs.
- It is a Permissioned Blockchain based DL;
- It implements a new system of transaction evaluation and registration, as explained in Section 2.2.4.A, that increases the speed of the Ledger to close to 2000 TPS, a great increase when compared to similar solutions [24];

[7] https://www.ansible.com/, accessed on 10th September 2021
[8] https://www.redhat.com/, accessed on 10th September 2021
[9] https://www.hyperledger.org/, accessed on 10th September 2021
[10] https://www.linuxfoundation.org/, accessed on 10th September 2021

- Allows for the usage of general purpose programming languages in the development of chaincode, such as Golang, Java, Javascript and Typescript;
- Utilizes LevelDB as a state database, that enables information to be stored in a fast accessing key-value store, ideal to store the assets as JavaScript Object Notation (JSON) dictionaries and with string keys, also providing immediate consistency, opposed to the more popular but worse eventual consistency;
- Since it is designed to be used as part of a bigger system, it has both an SDK and APIs in several languages that allow for interaction with the ledger integrated in other solutions like in our case.

These characteristics make Fabric the best choice for this solution, by giving the best performance and design characteristics that will improve the ability of our proof-of-concept tool to comply with all the requirements. However, it is important to keep in mind that the tool will be implemented in order to allow for the usage of different DL implementations (with the implementation of a module to interface with said DL).

3.3.3 Programming Language

Our solution will be entirely implemented in **Golang**[11]. This language was chosen due to several reasons:

- The usage of a single language decreases complexity of the system and eliminates the need for interfaces to exchange information between parts of the system implemented in different languages;
- The Hyperledger Fabric is implemented in Golang, and, because of that, the most well documented, complete and best performing API is the one in Golang;
- Golang is a recent programming language, with support for the newest software development trends such as library management;
- Although recent, Golang has a large number of libraries that implement key features needed for this solution such as a Web Server for serving the REST APIs, support for management of cryptographic material in a secure manner and concurrency mechanisms that enable multithreaded operations, important for the concurrent processing of requests;
- Golang is a compiled and strongly typed language, improving performance, that benchmarks show to be close to C and C++, and increasing security and reducing bugs due to compiler checks.

[11] https://golang.org/, accessed on 10th September 2021

4

Implementation

Contents

4.1 Development Methodology 29
4.2 Environment 29
4.3 Hyperledger Fabric 30
4.4 Modules Development 40

After the definition and specification of the general aspects of the solution in Chapter 3, we will now analyse in more detail the specific implementation of the tool, in accordance to the requirements and following the general architecture and data flows previously established. In this chapter we will first present the **methodology for the development**, then explain the **development/running environment** and finally go step by step over the **development process**, explaining the taken decisions over the development of the solution.

4.1 Development Methodology

For this project, we followed a **bottom-up approach** in development. After setting up the environment in which development and solution execution will take place, that we will present in section 4.2, we started setting up the ledger, since the solution we will develop will base all the main logic, both for business logic and user authentication, in the ledger, via **Smart Contracts**, and the storage in the **Ledger state database**. This way it made sense to setup those bases first and only then develop the tool's modules.

This methodology of development also enables us to **incrementally test the system** since each implemented part or feature will always have all its dependencies already implemented. This way we could ensure that the tested component was always working over other already tested components, reducing each test's scope. However, this methodology also presented some challenges because when implementing some feature we had to always plan the upcoming features to ensure that all future feature's implementation would not depend on something that was not thought before and not implemented. For example, if feature *X* depends on some subfeature *Y.a*, since *Y* was always developed first, we had to plan subfeature *Y.a* before it was actually needed. Since there were some minor changes during development, it was needed to revisit already implemented features and modules some times. For simplicity, in this document we will omit those minor setbacks and present the development as if those setbacks did not occur and present the development that lead to the final version of the solution.

4.2 Environment

The project was developed in and for an environment that aims to represent a common approach in the industry, by using **containers** to host different services, that run in **virtual machines**. Furthermore, this type of environment is very flexible, enabling us to make changes and apply them very easily.

To setup the environment, **Vagrant**, together with **VirtualBox**, were used. Vagrant enabled us to automate the provisioning and configuration of all assets related to this solution, given that scripts to do so were supplied. The central configuration file for the environment is called *Vagrantfile*, and it specifies the architecture of the infrastructure.

For this deployment, we choose to provision two virtual machines, one to run the Hyperledger Fabric components, and another to host and run our solution. Both machines are running Ubuntu 20.04 LTS, and have 4GB of Random Access Memory (RAM) and 4 CPU cores each. We choose 4GB of RAM since it was a manageable value for our VM host and both virtual machines were not constrained by it (not displaying high RAM usage), and we followed the same reasoning for the CPU core count. Files and folders are shared between the machines using shared folders between the machines and the host. Using this shared folder structure enabled us to easily emulate the physical distribution of files (mostly keys) from the CAs to the services using them.

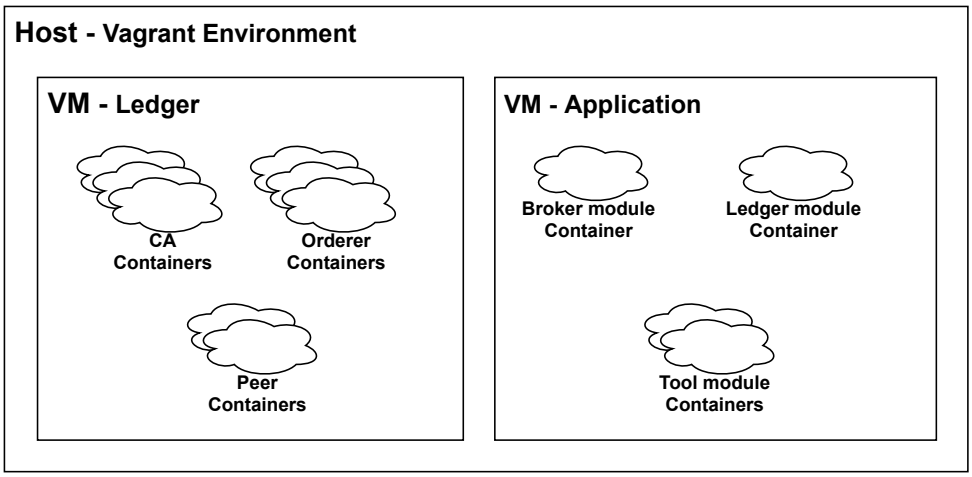


Figure 4.1: Host-VM-Container architecture

Each machine then has **docker** and **docker-compose** installed. The docker-compose system enables for the automatic setup of more complex docker container environments using YAML files. Each VM has a configuration file that specifies the architecture of all containers in that machine. The diagram in Figure 4.1 summarizes and schematizes this architecture.

There are then a large amount of scripts that specify the configuration, installation and compilation of both the Ledger and our developed code. All the instructions in those scripts will be discussed in the following sections, where we will explain how this tool is set up and developed, component by component.

4.3 Hyperledger Fabric

Since Hyperledger Fabric is an already existing solution, for this project, no code related to Fabric development was produced. However, it was needed to configure an implementation of Hyperledger Fabric to use as a DL base of this solution. In the next sections we will present the configuration steps for the Ledger, along with the justifications for some decisions taken in said configuration and deployment.

4.3.1 Certificate Authorities

Since Hyperledger Fabric is a permissioned ledger, **every participant in the Ledger must be authenticated**. For this authentication to be secure and cryptographically sound, the best method of identification is via the generation, for each entity of an identity based on a key pair, together with the corresponding certificate, that must be signed by some trusted identity. The identity that has the power to sign such certificates is called a CA. They are the first components of the Hyperledger Fabric system to be deployed.

For our solution we will use the **Fabric CA**, an implementation of a CA created by the Fabric project, since it is already tuned to generate the correct cryptograpic material that the Fabric system consumes, as previously explained in Section 2.2.4.A. We will deploy **three instances of CAs**. Since it is recommended to have, at minimum, one organization for the Orderer nodes and another for the Peer nodes, we will name those organizations **Org0** and **Org1**, respectively. It is important to understand that this recommendation is just to simplify the deployment, as stated in the documentation, and it is possible to have any number of organizations with any combination of nodes. There will be a third CA instance that will act as a Transport Layer Security (TLS) CA. There are two types of certificate usage in Fabric, one is for **identification of an identity over the Ledger**, while the other usage is for **authentication of the communication between nodes**. The two Org0 and Org1 CAs are responsible for the emission of certificates for each Organization's Identities Identification. The TLS CA will then be responsible for the emission of certificates that will be used to secure TLS communication between the nodes.

It then makes sense that each user will only have a Org0 or Org1 certificate, while nodes (Orderers and Peers) will need to have both a certificate from its Organization's CA and another from the TLS CA.

From a security perspective, the root certificates of the CAs will be **self-signed** since the tool will be only used as a proof-of-concept, but it is possible, and recommended, that in a real deployment the certificates be signed by a trusted external CA, forming a certificate chain that follows back to a trusted root CA.

From a deployment perspective, all the CAs are deployed as containers on the Ledger VM.

4.3.2 Identity Generation

After the deployment of all the CAs, the necessary basic identities are created. The only identities already created are the CA bootstrap identities, only used to initialize and to provide authentication to the creation of the new identities that will be used throughout the whole system.

The identity creation process is straightforward, and using a provided tool, called *fabric-ca-client*, that is responsible to communicate with the CAs and both creating the users identities and retrieving their certificates.

The first step is then to **register the user on the CA**, a process similar to creating an user account. At this moment, we supply the **username** (that will be part of the Common Name in the certificate), a **password** and the **type** of user we are creating (that can be admin, user, orderer or peer). It must be noted that an admin user in this scenario is a Ledger admin user, that can configure nodes or its organization, it must not be confused to an attribute of ”admin” as part of any solution that is using Hyperledger Fabric as its Ledger, such as ours. This process only creates the user identity, not generating any cryptographic material yet.

The next step is to then generate the certificate and keys that identify the identity in a process called **Enroll**. Similarly to the register process described earlier, it is done using the *fabric-ca-client*, this time authentication with the username and password specified earlier. This is the only usage for said password. The tool will then create an MSP folder with all cryptographic material that identifies the identity over some CA.

Listing 4.1 exemplifies the process of registering and enrolling an identity, in this case a Peer for Org1.

Listing 4.1: Registration and Enrollment of an Identity

```
1 ./fabric-ca-client register --id.name peer1-org1 --id.secret peer1PW
2     --id.type peer -u https://tls-ca:7054
3 ./fabric-ca-client enroll -u https://peer1-org1:peer1PW@org1-ca:7054
```

4.3.3 Peer Configuration

After the creation of the identities, the next step is to configure and deploy the peers. The peers are responsible for the **execution of the smart contracts**, using one container for each installed smart contract.

In the previous step, identities for two peers were created. In our implementation we choose to deploy two peers due to the documentation’s recommendation for redundancy. Although the peers communicate between themselves, in a process called *gossip*, they don’t make any type of consensus or synchronization, so the recommendation for a minimum of two peers is purely for redundancy in case of failures. By having two peers we also prove that the system is scalable for larger numbers of peers. In our implementation both peers belong to Org1, however, it is possible for peers from different organizations to work together in the same channel, with the same smart contracts. In larger implementations with large numbers of smart contracts, it is also possible to install those smart contracts so that not all peers are responsible for the execution of all smart contracts.

The configuration of the peers is simple, just requiring access to the generated MSPs of the peers.

Since the peers are nodes that will handle communication, each peer has two MSPs, one coming from the Org1's CA, with the identity of the peer as belonging to that organization, and another generated by the TLS CA, with the purpose to secure the TLS secured communications of the peer.

Since each peer will run the smart contracts in containers, it is also needed to provide the peer with access to the Docker's socket, so that the peer can create, delete, and modify the needed containers to run the smart contracts. The peers are also deployed inside containers.

4.3.4 Orderer Configuration

Similarly to the peers, after having the needed identities created, the orderers were configured and deployed. The Orderers are responsible for maintaining the blockchain across all of them. For this, they receive the **approved executions of smart contracts** from the peers and **generate the blocks that will be appended to the blockchain**. Since the blockchain must be equal across all peers, they run a **consensus algorithm**, that, together with a deterministic algorithm to order the blocks to be appended, ensure the uniformity of the chain across all nodes. They are also responsible for handling the reading of the chain if it is needed, either by request from the peers or by a user directly using the orderer command line. Since the orderers run a consensus algorithm across them, it is recommended to have at least three orderers, so that, even if one of them fails, the remaining two can still make decisions. It is also possible to only have one orderer but that configuration is not recommended.

The configuration for the orderers is similar to the peers' configuration, also requiring an identity from an organization, in our case that organization is Org0 for all orderers, and a MSP from the TLS CA to enable secure communications between the nodes. The orderers are also deployed in containers, requiring persistent storage to store the blockchain.

4.3.5 Channel Creation

After having all the nodes configured and deployed, the next step is to create a Channel. A channel is effectively an instance of a blockchain, that supports a policy of access, together with the corresponding smart contracts.

In order to start the blockchain it is needed to generate its first block, or **genesis block**. The genesis block is where the general configuration of the channel is stored. Since after it is added to the blockchain it becomes immutable, if there is a need to change the configuration, an update block is generated.

4.3.5.A Genesis Block

To generate the genesis block, a specific tool, *configtxgen* is used. This tool will read a configuration file, usually called *configtx.yaml*, is a YAML file that specifies all the policies related to the channel it will

start.

It is divided in several sections, the first being organizations. In this section, all organizations that will participate in the channel are registered, together with their nodes, and policies. In our case, we only create one channel, where the two organizations, Org0 and Org1, participate. There are four policies, that in our case are configured identically for both organizations:

- **Readers** - This is the policy for reading the chain, in this implementation only Admins, Peers and Clients can read the chain. Clients are in fact the users of the chain, they are required to have read access to the chain in order to commit changes (after being always approved via smart contracts).
- **Writers** - This is the policy for writing new blocks to the chain. Only Admins and Clients are able to do so.
- **Admins** - This policy refers to administrative actions over the channel, such as configuration changes.
- **Endorsement** - The endorsement is made by the peers. This is the process used by the peers to certify that a modification registered in some block is accepted by the peers, after running the corresponding smart contract. As such, only peers have permission for endorsement.

In this section, we also register the nodes that will take part in the channel, by organization. In our implementation, from Org0 we registered the three Orderers, and from Org1, we registered the two Peers.

The next relevant section is the Orderer section, in which the type of consensus used is defined. In this solution we are using the **Raft algorithm** since it is the most recent and capable one, and recommended by the documentation for new deployments that don't have to maintain retro compatibility.

The next two sections, Channel and Profiles are where the configurations are organized in a structure. In our case, we also had to specify the number of members of each policy is needed for the request to be approved. For both Read and Write, it will require only one identity from the authorized group. However, to execute administrative tasks, a **majority** of the Admins is needed.

4.3.6 Chaincode Development

After the setup of the channel, the Ledger is in fact working. However, to make transactions, Smart Contracts, or Chaincode, needed to be implemented. In our case, only one **Smart Contract is needed, that will handle all asset modifications**. Before the implementation of any logic, it was needed to create specifications for the storage of the information. With this in mind, two types of **Assets** were created:

- **Asset** - Represents any asset in the infrastructure, such as Servers, VMs, Containers, and others.
- **Applied Tools** - This data structure stores information about each and every tool execution against the infrastructure, such as the execution of an Ansible Playbook.

These data structures were coded to be as general as possible while retaining all the needed information about each asset. This way we achieve the goal of **expandability** with no need for code refactoring. Since the developed tool is to be used as a proof-of-concept, only key values, representative of the possible information to be stored were indeed stored and later processed in the Smart Contract.

Listing 4.2: Asset Structure

```
type Asset struct {
    ID              string                  `json:"id"`
    Type            string                  `json:"type"`
    Location        string                  `json:"location"`
    Owner           string                  `json:"owner"`
    SpecRamGB       int                     `json:"spec_ram_gb"`
    SpecCpuCores    int                     `json:"spec_cpu_cores"`
    AppliedTools    []string                `json:"applied_tools"`
    IpAddrs         []string                `json:"ip_addrs"`
    Dependencies    []*DependencyRelation   `json:"dependencies"`
    Dependants      []*DependantRelation    `json:"dependants"`
    Implemented     bool                    `json:"implemented"`
}
```

In Listing 4.2 we can observe the structure for a typical infrastructure asset, as it is implemented in Golang. For storage and communication purposes, it is directly converted to JSON format. Each field has a specific purpose:

- *Location*, *SpecRamGB*, *SpecCPUCores* and *IPAddrs* - These will not be used in any type of verifications in the Smart Contract and are just present as an example of different types of asset related data storage;
- *ID* - Unique ID for identification the asset;
- *Type* - Identification of the type of the asset (e.g., Server, VM, Switch, Cluster);
- *Owner* - Owner identification, used for permission verification;
- *AppliedTools* - List of Identifiers of the Applied Tools structures that represent tool executions that affected this asset;

- *Dependencies* and *Dependants* - Lists of pairs (Asset ID, Applied Tool ID) that represent all the dependencies and dependants of the Asset, together with their origin.

Listing 4.3: Asset Structure

```
type AppliedTool struct {
    ID                  string                  `json:"id"`
    AppliedTo           []string                `json:"applied_to"`
    AssocDependencies   []*DependencyRelation   `json:"assoc_dependencies"`
    ToolName            string                  `json:"tool_name"`
    FileName            string                  `json:"file_name"`
    FileHash            string                  `json:"file_hash"`
    FinalState          *State                  `json:"final_state"`
    Reverted            string                  `json:"reverted"`
}
```

In Listing 4.3 we present the structure for the registry of the execution of some infrastructure management tool. Similarly to the Asset structure, this structure is also implemented to be as general as possible, so that it can be used with any tool. The data fields of this structure are all used in the Smart Contract and in our application:

- *ID* - Unique ID for the identification of the tool execution;
- *AppliedTo* - List of identificators of the Assets affected by the execution;
- *AssociatedDependencies* - List of new dependencies introduced by this execution;
- *ToolName* - Identificator of the used infrastructure management tool;
- *FileName* and *FileHash* - Used to identify the files used as input to the tool, that are stored on the Tool's modules;
- *Final State* - Stores the final state reported by the tool, used to check for failures, successes and modifications to the infrastructure;
- *Reverted* - Indicator of whether the execution was reverted or not;

After the definition of these basic and universal data structures, the development of the Smart Contract started. The Smart Contract has several sections, that we will now analyze one by one.

4.3.6.A Asset Type Management

As stated previously, the Assets must have a type. To allow for the dynamic management of types, those are not directly coded in the Smart Contract, instead having a mechanism for adition and removal of types. For this mechanism to work, a special asset in the ledger was created, *TypeTracker*, that is in fact just a list of the existing and allowed Asset types. This structure is created when the contract is initialized, via its function *InitLedger()*.

For the mechanism to work, functions to Get, Add and Remove Asset Types were implemented. Although any user can Get the list of Asset Types, only Admins can Add or Remove Asset Types. This introduces the first form of **permission management**, where we verify if the user making the request has a specific identifier in its identity certificate, thus in fact implementing a **Role or Attribute Based Access Control**. Those identifiers can be managed when registering a user on the CA. **On the Smart Contract, the verification is trivial** since the Hyperledger Fabric Smart Contract API exposes the calling Client Identity within the Transaction Context that is provided as argument to every function in the contract.

4.3.6.B Asset Management

After the introduction of the Asset Types, the Assets can be coded in the Smart Contract. The Assets have four main associated functions, to **Get**, **Register** a new Asset, **Modify** an existing Asset, and **Remove** an Asset. From a logic and coding perspective, the action of getting information about an asset is simple. However, we have to have in mind the need for privacy and that a **user can only see its Assets**. In chaincode, this verification is very easy due to the presence of the ID of the owner of some Asset in the Asset's data and the possibility to get the ID of the client calling the Smart Contract's function from the Hyperledger Fabric Smart Contract API that exposes the calling Client Identity within the Transaction Context that is provided as argument to every function in the contract. The code for the get asset function is exemplified in Listing A.3.

From a business logic perspective, the registration of a new Asset is more complex since it is needed to **verify** that all essential data fields about the Asset are present, that the Asset Type is valid and it is needed to **parse the dependency list**. By default, an Asset is created without any dependencies or dependants. However, it is possible to specify dependencies for that Asset. When the Asset is registered, the logic present in the Smart Contract will automatically parse those dependencies, **check if they are possible** (e.g., the Asset from which the new Asset depends exists) and automatically add the newly registered Asset to the Dependants List of the Asset the new Asset depends on. It also verifies that the new Asset has a maximum of one Applied tool, since the Asset can only be created once and by only one tool, as it can be seen from the sample code in Listing A.1

On the opposite side, the removal of an Asset is simple, with the only verifications to be carried being the check for **permissions** by the calling client (that must be the owner of the asset or an admin) and

that the Asset to be removed **doesn't have any Dependants**, as exemplified in Listing A.2.

The remaining action is the modification of an Asset. In this action several constraints are checked: The permission of the calling client to modify the Asset, and that the basic Asset identifying information is no changed (such as the ID, the Type and Owner). The removal of applied tools is also not possible, because it must be done using the proper Applied Tool removal method. Any dependency changing is parsed and verified for the conditions both to removal and addition, as specified for the creation of a new Asset.

4.3.6.C Dependency Management

Although **dependencies** are properties and **relations between Assets**, and can be created by Applied Tools, they also specific public functions to provide the functionality of manual creation and deletion of dependencies without the need for the explicit modification of an Asset. Both the functions to create and to delete dependency relations verify the permissions of the user to do so, by comparing the owner of the Assets that will form the dependency relation with the calling user. For the dependency to be created or removed, the user identifiers must match, which signifies that the user calling the functions is the **owner of both Assets**.

4.3.6.D Applied Tool Management

The management of the Applied Tools is done in a similar manner to the one applied to the Assets. However, due to the context, the functions that will be presented will be done to handle several actions:

- **Creation of new Applied Tools** - this creation will only represent that the Applied tool will run, and represents that a tool is currently running.
- **Finish of the Applied Tool** - this action is what confirms the termination of the execution of said tool, and, depending on the success of the execution, will change the affected Assets, adding the Applied Tool data structure ID to the list of applied tools present on each Asset, and creating the dependencies specified by the tool (such dependencies are already specified in the Applied Tool data structure, and the Smart Contract is only responsible for the distribution of the information about the dependency across the affected Assets).
- **Reversion of the Applied Tool** - this action is responsible for the removal of the dependencies introduced by the Applied Tool, and will mark the Applied Tool as reverted, for documentation purposes.
- **Getting an Applied Tool** - this function is responsible for the handling the user requests for information about a specific Applied Tool. A user can only get such information if any asset of it was

affected by this Applied Tool.

It is important to be noted that the Smart Contract receives information about the Assets, Dependencies and Applied Tools in an already normalized format, ready to be stored, being only responsible for the verifications of validity of data (for example, that all supplied IDs are valid, that the Asset Type exists), permission verification varying with the function to be executed (for example, it can check for the ownership of Assets, verify if the calling user is an admin) and correct storage of the received information (for example, when receiving an Applied Tool, all new dependencies must be propagated to all involved Assets). The conversion of the information to these uniformed types, that will be used across the whole solution is a responsibility of the creator of said information, being it the user or any of the tools modules after the execution of said tools. It is also important to note that since this solution is to be used as proof-of-concept, the set of verifications implemented is not exhaustive, and it is to be understood as a significative representation of the capabilities of the Smart Contracts, both in permission management (where more roles and attributes can be easily added, with more complex verifications) and business logic verification (where more aspects about Assets could be tracked and processed).

4.3.7 Chaincode Installation

After the coding of the Smart Contract, it is needed to submit it to the ledger so that it becomes available. This step is usually called Chaincode Installation. The whole process is well documented in the Hyperledger Fabric documentation:

- Compilation, since the Smart Contract is coded in Golang;
- Packaging into a universal format used by Fabric, using the peer CLI, supplying the executable location, programming language and a name for the Smart Contract;
- Installation on every peer that will run requests for the Smart Contract. In our case, both Peers will have it installed;
- Authenticated as admin, approval of the Smart Contract for the channel;
- Still authenticated as admin, commit of the Smart Contract to the chain, making it usable;
- Manually invoke the *init* function of the Smart Contract in order to initialize any needed state.

Listing 4.4: Installation of a Smart Contract

```
1 peer lifecycle chaincode package inventoryMgt.tar.gz
2     --path ./inventory-management/
```

```
    --lang golang --label inventoryMgt_1.0

peer lifecycle chaincode install inventoryMgt.tar.gz

peer lifecycle chaincode approveformyorg -o orderer1-org0:7050
    --channelID channel1 --name inventoryMgt --version 1.0
    --package-id $CC_PACKAGE_ID --sequence 1 --tls
    --cafile tls-cert.pem

peer lifecycle chaincode commit -o orderer1-org0:7050
    --channelID channel1 --name inventoryMgt --version 1.0
    --sequence 1 --tls --cafile tls-cert.pem
    --peerAddresses peer1-org1:7051 --tlsRootCertFiles tls.pem

peer chaincode invoke -o orderer1-org0:7050 --tls --cafile tls-cert.pem
    -C channel1 -n inventoryMgt --peerAddresses peer1-org1:7051
    --tlsRootCertFiles tl.pem -c '{"function":"InitLedger","Args":[]}'
```

Listing 4.4 exemplifies, for a single Smart Contract and a single Peer, the process of installation of chaincode, as done in our configuration. It must be noted that the approval and commit processes, while done using the Peer CLI are in fact done over the Orderers since they are the ones that manage the aspects related to the administration of the channel and chaincode.

Having the Ledger running and serving requests, the focus then turned to the development of the modular solution that will harness the powers of the Ledger, and the invoke the logic retained in the Smart Contract, as specified in Section 4.3.6.

4.4 Modules Development

As exposed before in Chapter 3, our solution is a developed set of modules that harness the powers of the DL to enable for easier and more secure management of IT infrastructures. Following the development methodology explained in Section 4.1, we start with the Ledger module, since it will only depend on the Ledger, that is already implemented and deployed, and will be needed by all the other modules, either directly, such as the Broker module, or indirectly, such as the Tools modules that will depend on the Broker, followed by the Broker module, leaving the tools modules for last. However, before presenting any specific modules, there are some module design considerations to address.

4.4.1 Module Architecture

Although every module has a different purpose within the application, we followed a general architecture for each module, that simplifies the data flows inside the modules and is easier to code, leading to less mistakes or poorly implemented data paths.

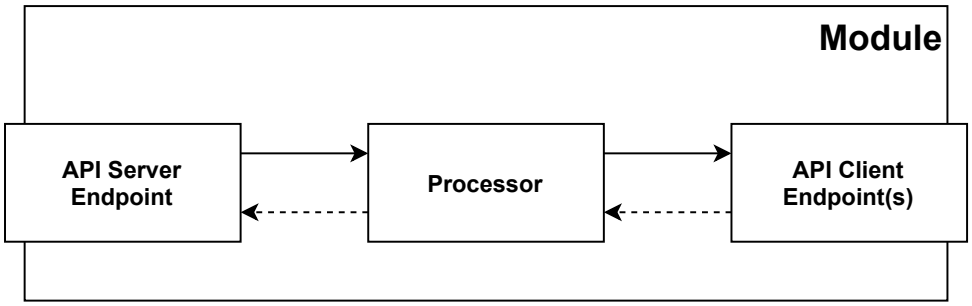


Figure 4.2: Module Architecture and Data Flow

As presented on Figure 4.2, each module has three basic components: API Server endpoint, Processor and API Client Endpoint. The API Server Endpoint component is responsible for serving the public API of the module and handling requests made to it. Since all the APIs in this project are REST APIs, this server module is in fact a Hypertext Transfer Protocol Secure (HTTPS) web server, using the Gin-Gonic Framework[1], that enables us to set the endpoints addresses and an handler function for that specific address. In our implementation, the handler function will parse the received information, being it via URL encoded values or as the Body of a POST request, and marshall it to already defined structures in Golang, to enable further processment. The information will then be sent to the Processor component, that is responsible of applying any needed logic (that is module and request specific), and can automatically return some information to the API Server module to be returned to the calling client, or request more information from other modules.

To request information from other modules, a third component is implemented: the API Client Endpoint. This component is responsible to make API calls to other modules. It receives a request from the Processor component, converts any needed information to JSON, so that it can be sent easily in a body of a request, and makes an HTTPS request to the API Server of other module. It must be noted that a module can have more that one API Client Endpoint if it is needed to make requests to more that one other module. To keep the code organized, it was decided to restrict each API Client component to connect to only one type of module. Although for simplicity the Client component has been called API Client Endpoint, it can serve other client-like functions, such as being a command line interface, or the client for the file system (that provides abstraction over the command line/ file system activities).

[1] https://github.com/gin-gonic/gin, accessed on 19th September 2021

4.4.2 Ledger Module

The Ledger Module is the first module to be implemented. This module is responsible for providing an abstraction layer over the Ledger, by receiving the requests as a REST API, as used by all other modules, and converting them in requests to Hyperledger Fabric, using the Fabric Go Software Development Kit (SDK). The API Server implementation is simple, reflecting the methods present in the Smart Contract, since this module does not implement any logic besides the minimum needed for the conversion of the requests. Since the use of this API is purely internal to the tool and just reflects the functions present on the Smart Contract, its specification will not be presented here in full. The only addition to the API that is not present in chaincode is the *login* endpoint, that receives the user credentials, together with the user's Session ID, that are cached on the module, and used to identify following requests made by the user and to make the authentication of each request over the Ledger.

In order to make a transaction on the Ledger (in other words, running a Smart Contract function), it is necessary to use the Fabric SDK. In order to establish a connection to the Ledger, firstly it is needed to create a connection configuration file, in YAML format, that contains the addresses of the Orderers and Peers of the network, along with the root certificates in order to enable the SDK to avoid possible man-in-the-middle attacks by being able to reconstitute the certificate chain received from each of the nodes up to the provided root certificates that are assumed to be trustworthy.

After writing the configuration file, it is possible to, in code, create a connection to the Ledger. Firstly, it is needed to create a wallet to store the received user credentials and to send them in a secure way to the Ledger to be verified. Using a gateway object (provided by the SDK) and providing the wallet and the configuration file written before we start a connection to the Ledger. After the establishment of said connection, we must specify the Channel we are connecting to, and then the Smart Contract we want to call functions from. It must be noted that the connection can be reused for different Smart Contract invocations, even in different Channels. This reuse is encouraged because it avoids the delays and processing associated with an entirely new connection. After the specification of both the Channel and Smart Contract it is then possible to call functions from the Smart Contract. The Fabric SDK provides two different ways of doing so: Submission and Evaluation, the difference between the two being whether the results are stored in the Ledger or not. We can thing of an Evaluation as a dry run of the Smart Contract, that can be useful to verify the possibility of making some change without effectively making that change, since the process is similar, apart from the effective writing to the Ledger state database and blockchain. In both methods, it is possible to send information along the request as parameters and to receive information from the Smart Contract as the return values.

Listing 4.5: Fabric SDK Connection

```
Wallet.Put(sessId, identity)
gw, _ := gateway.Connect(
        gateway.WithConfig(config.FromFile(configPath)),
        gateway.WithIdentity(Wallet, sessId),
    )
network, _ := gw.GetNetwork("channel1")
contract := network.GetContract("inventoryMgt")

result, _ := contract.EvaluateTransaction("GetAssetTypes")
```

Listing 4.5 exemplifies the connection process from the Fabric SDK to the Ledger, as explained in the paragraph before. It must be noted that all errors are ignored in this example for simplicity. In this case, a connection is made to the Ledger in the name of the user with Session ID *sessID* and credentials stored in *identity*, specifying then the Channel as *channel1* and the Contract as *inventoryMgt*. Then an Evaluation is made for the method *GetAssetTypes*. In this case an Evaluation is made (opposed to a Submission) because the called method is a ”read” method that will never change the state of the Ledger. This way it is possible to save time on read methods since it avoids the whole process of committing information.

The Ledger Module work and data flow can then be resumed as receiving requests over its API Server, parsing and marshalling that information over to Golang data structures, calling the corresponding method on the Ledger using its SDK, and then, after receiving the result, convert it back to JSON, and send it as response to the original request.

4.4.3 Broker Module

After the implementation of the Ledger Module, the Broker module is implemented. Although according to the development methodology presented in Section 4.1, we should firstly develop the Tools Modules, since the Broker Module also depends on them, we chose to implement the Broker Module firstly because it connects to the Ledger module and we wanted to ensure it was working as intended, specially the user session feature, that must coexist in both the Broker and Ledger modules.

The Broker module works as a center piece in this solution, being a sort of work distributing module, and also the user endpoint for making requests. Since the Ledger Module was already implemented, we started by implementing the Ledger Client Component (following the general architecture presented in Section 4.4.1). The implementation of this component is basically the reverse process of the Server Component of the Ledger Module: for each endpoint in the Server Component we create a function in the Client that is responsible for the conversion of the Golang data structures used inside the module to JSON and makes an HTTPS request to the Ledger Module. It must be noted that for this, the Broker

had to store the connected users' session IDs and send them along the requests as a Cookie in order to enable the Ledger Module to distinguish between users.

After the implementation of the Ledger Client component, we decided to implement the API Server component, creating the specification for the user accessible API Endpoint to control the tool.

Table 4.1: Broker Module public API

Endpoint	Method	Description	Receives	Responds
/login	POST	Handles user login	Zip file containing credentials	Session ID Cookie
/logout	POST	Handles user logout	Session ID Cookie	-
/assets/types	GET	Get Asset Types	-	List of Asset Types
/assets/types	POST	Add Asset Type	New Asset Type	List of Asset Types
/assets/types	DELETE	Remove Asset Type	Asset Type to be removed	List of Asset Types
/assets/{type}	GET	Get assets by type	Asset Type	List of Assets
/asset/{id}	GET	Get asset by id	Asset ID	Asset
/asset	POST	Add new asset	Asset	The added asset with its generated ID
/asset/modify	POST	Modify existing asset	Modified Asset	-
/asset/{id}	DELETE	Delete Asset	Asset ID	-
/asset/{assetId}/dependencies/{dependencyId}	POST	Adds a dependency where *assetId* depends on *dependencyId*	Asset and dependency IDs	-
/asset/{assetId}/dependencies/{dependencyId}	DELETE	Removes a dependency where *assetId* depends on *dependencyId*	Asset and dependency IDs	-
/tools/{toolName}/plan	POST	Simulates tool execution and returns changes	Tool name and Zip file containing tool plans	Applied Tool
/tools/{toolName}/execute	POST	Starts tool execution and returns changes for final approval	Tool name and Zip file containing tool plans	Applied Tool with generated ID
/tools/{toolName}/{appliedToolId}/confirm	POST	Approves tool execution, makes tool execute	Tool name and Applied Tool ID	Applied Tool with final status
/tools/{toolName}/{appliedToolId}	GET	Gets information about a previous tool execution	Tool name and Applied Tool ID	Applied Tool with final status

Table 4.1 summarizes the exposed API. We can divide this API in three main sections: **Login**, **Asset Management** and **Tool Management**. The **Login** section provides two methods for login and logout of the user, necessary due to the session based nature of the solution. To make a login, the user must submit the zip file containing its Certificate and Key, generated by the user's organization CA, that will be evaluated by the ledger, and, if correct, will generate a unique Session ID, served as a Cookie, that must be present in all calls to the API in order to authenticate and authorize the user.

The **Asset Management** section provides methods for the creation, deletion and modification of assets and their dependencies. It must be noted that all asset information sent and received is done via a JSON representation of the Asset, equal in format to the one presented in Section 4.3.6, and that the ID of the Asset is always generated by the Ledger when registering the Asset, and when creating an Asset, if the ID field is populated by the user manually, it will be ignored. All the Asset related requests are forwarded directly to the Ledger, where they will be processed according to the Smart Contract rules.

The **Tool Management** section provides the necessary methods for the automatic invocation of the infrastructure management tools, that will lead to the automatic detection of changes and subsequent modification of the state database. Both the Plan and Execute methods receive a zip file containing the tool specific plans, already written and formatted in the tool's specific language. The main difference between those two methods is that the Plan method only runs the tool and the smart contract processing in dry mode, not making any changes to the infrastructure or state database, but making all the verifications either done by the tool or the smart contract, and returning the result of those verifications to the

user. On the other hand, the execute method will trigger the full tool execution process as specified in Section 3.2.2.C, that will return to the user the same results as the plan method would, however, providing a tool execution ID, that the user can then confirm using the Confirm method, making the changes to the infrastructure and state database permanent.

After the implementation of the API Server Component, the Processor Component is implemented. For the methods concerning Asset Management, the processor does not make any actions or verifications, just forwarding them to the Ledger module. However, for the Tool management methods, the logic specified both in the last paragraph and in Section 3.2.2.C was implemented, making the broker responsible for the several requests made to both the Ledger and Tool modules.

The last component of the broker module to be implemented is the Tools Client component. This component, similarly to the other Client components, is responsible for the conversion of the internal representation of the information to JSON and for making requests to the specific API Server, in this case, the API Servers of the various tool modules. The only particularity of this component was that, since every tool has a module, there are in fact different addresses that this component may need to connect. We chose to make a single Tool Client component to enable the addition of more tool modules without the need to implement more client components in the Broker module. This client component reads a configuration file where all the tools are specified, along with their modules' addresses. When the component is triggered to make a request to some tool, it will read the file, and get the correct address to the specific tool, making the request to that address. This enables the Broker module to support new tool modules even without a reboot.

4.4.4 Tools Modules

The last modules to be implemented are the tool modules. These modules, one per tool, are responsible for the receival of the requests from the Broker Module, and to convert them in executions of the corresponding tool using a shell emulator, and then to convert the output of the tools to the standard formats for Assets and Applied Actions used in out solution, as specified in Section 4.3.6.

The modules follow the same general module architecture, having an API Server component, that implements the handling of the requests that the Broker may execute, and that converts the received data to the standard formats used inside the module. The requests are then passed over to the Processor component that is responsible for the usage of the Client components to fulfil the request. However, in the tools modules the Client components side, instead of providing connections to other APIs, provide abstraction layers over the file system and the shell. Since the tool modules will need to store the executed plans (since the Ledger will only store hashes of those files) and the tools consume their plans from the file system, it is needed to organize and place them in the correct folders. A file system client module provides a set of functions that receive and return in memory copies of the files, taking care of

their storage in the file system. The shell abstraction module, in its turn, is responsible for the calling and execution of the tools, via a emulated shell, and then the parsing of the output of the tools in order to detect changes to the infrastructure, being responsible to convert the output of the tool into a set of new and/or modified Assets and a Applied Tool, with the correct fields correctly filled.

This way, when receiving a request, the processor can then call the file system component in order to store the received plans in the plans archive and to create the execution folder for the execution of the corresponding tool. Then, it calls the shell emulator component in order to effectively execute the tool and parse its output into the default data structures used in our tool. The processor then calls the file system component again to do a cleanup of the working directory of the tool, while saving in the plans archive all relevant files. The processor can then return to the Server component all the information about the execution, that will be sent back to the Broker module for further processing and storage in the Ledger.

However, it is important to note that in the implemented solution, the dependency creation must be done manually since it is not possible, from the general Terraform or Ansible plans, to detect dependencies between Assets. The only dependencies that can be detected are the ones involving Terraform provisioning VMs in a cluster or host. In that case, if the cluster or host is present in the inventory, a dependency relation will be created.

5

Evaluation and Result Analysis

Contents

5.1 Test Scenarios 49
5.2 Scenario 1 - Authentication 50
5.3 Scenario 2 - Normal Workflow 51
5.4 Scenario 3 - Authorization 57
5.5 Scenario 4 - Dependency processing 58
5.6 Scenario 5 - Rollback of tool applied actions 60
5.7 Load and throughput 62
5.8 Result Discussion 66

The focus of this dissertation is the evaluation of the possibility of harnessing the capabilities of a Blockchain based DL to improve the security and automate constraint verification in the management and logging of actions taken over an IT Infrastructure. To enable for this evaluation, a proof-of-concept tool was developed. This tool can be divided in two main components: The Ledger and the user and infrastructure management tool interface. While the ledger used, Hyperledger Fabric, is a already developed solution, we had to correctly configure it for our use scenario and develop the Smart Contracts, or chaincode, that controls all the information that is written in the ledger, along all the associated permission and business constraints verifications, thus turning the ledger into the processing center of the tool. The user and tools interface was entirely coded by us, in order to provide an interface for the user to communicate with the ledger and also to integrate the tools into this solution by enabling the user to trigger tool execution that is automatically registered and processed by the logic present in the Smart Contracts.

In this chapter, we will present a qualitative and quantitative evaluation of the developed solution, resorting to the creation of different scenarios that can both evaluate the tool characteristics, according to the previously set requirements, and represent real world and industry representative use cases. After the presentation of the results obtained in each of the scenarios we will discuss those results. The scenarios will be simulated, with the tool running in the same environment as used for the development, utilizing an host supporting two VMs that run several containers for the execution of the several components of the tool. This environment is further detailed in Section 4.2. The target infrastructure is also simulated via the usage of an OpenStack[1] environment deployed on several VMs running in a physical server. Although this will result in nested virtualization, that can negatively impact performance, this factor can be ignored since there will be no performance evaluation of this simulated infrastructure.

5.1 Test Scenarios

As briefly explained in the paragraph above, in order to evaluate the developed tool, we setup different usage scenarios that aim to test different aspects of the tool, as well as exemplify its usage, at the same time trying to give a representation of realistic scenarios according to possible industry usage for this tool. Each simulated scenario will evaluate or prove a particular functionality of the developed solution, usually via a qualitative analysis of several aspects, but also via quantitative analysis of some key metrics of our solution. The evaluated aspects will be presented and explained for each scenario. Since the user interaction with the tool is done via a REST API, served by the Broker Module, we use cURL[2] to generate and send the requests and receive their output. Since cURL is accessed via a CLI, it is easy to setup the repetition of requests and their verification using Bash scripts.

[1] https://www.openstack.org/, accessed on 24th September 2021
[2] https://curl.se/, accessed 22nd September 2021

5.2 Scenario 1 - Authentication

The first scenario to be presented has as its objective to demonstrate and test the Login process. This process, as defined and explained in Chapter 4, consists in the execution of a login request, carrying a Zip file containing the user's credentials, that are evaluated by the Ledger, returning a Session ID as a cookie if the supplied credentials are correct. If the credentials are incorrect, the system will return an HTTP return code 401 - Unauthorized.

Listing 5.1: Successful Login Request

```
1
2  wigu@DESKTOP:~$ curl  -i --location \
3      --request POST 'https://broker-module:8080/login'\
4      --form 'identityFile=@"user.zip"'
5
6  HTTP/2 200
7  content-type: text/plain; charset=utf-8
8  set-cookie: SESSION_ID=JrV6hAzNN_w66NMWQJbp5tE4MwtH_GdC65Etr60MeCEQQ%3D%3D;\
9      Path=/; Domain=broker-module; Max-Age=86400; HttpOnly; Secure
10 content-length: 11
11 date: Sat, 25 Sep 2021 16:52:02 GMT
12
13 Logged in!
```

Listing 5.2: Unsuccessful Login Request

```
1  wigu@DESKTOP:~$ curl  -i --location \
2  > --request POST 'https://broker-module:8080/login'\
3  > --form 'identityFile=@"user_deny.zip"'
4
5  HTTP/2 401
6  content-type: text/plain; charset=utf-8
7  content-length: 20
8  date: Sat, 25 Sep 2021 16:52:15 GMT
9
10 Invalid credentials
```

As shown on Listing 5.1, when requesting a login for a user with correct credentials, the system appends to the header of the returning packet the *set-cookie* entry with the Session ID for that user, that

must be present on any upcoming requests. On the other side, when supplied with incorrect credentials, as seen on Listing 5.2, the system returns the 401 status code, indicating the login was unsuccessful. It is worth to note that both login attempts are registered on the ledger. It is important to note that the process of obtaining credentials via brute force is unfeasible since, while it is easy to generate a public and private key pair and generate a user certificate with the necessary information, it is unfeasible to generate the needed certificate signature, since the certificate must be signed by the CA, using its private key, that is never distributed. Any existing user certificate modification, for example, by a user adding an admin attribute to its certificate, is also unfeasible since the existing signature would then be invalid and the certificate must again be signed by the CA.

Also, if a user tries to use a invalid Session ID (either an non existent or an expired one) in any request, the system returns with a 403 - Forbidden status code, indicating that the user must login again.

All following scenarios will skip the login process, and it is to be assumed that all requests are accompanied by a valid session ID in order to be accepted, to improve clarity.

5.3 Scenario 2 - Normal Workflow

The second scenario is dedicated to the demonstration of the capabilities of the tool for the support of a normal infrastructure management workflow, without considering failures. In this scenario we start the tool with a blank infrastructure, with no assets registered in the ledger. We will then manually create a series of Assets, with the purpose of demonstrating both the capability of manually adding Assets and populating the Ledger to provide a more realistic initial ledger state for the remaining actions to be carried out. We then modify and delete some Assets. After the demonstration of the manual Asset management capabilities, we will trigger the execution of actions using both implemented tools, Terraform and Ansible, to provision and automatically register new Assets in the Ledger, and to make configuration changes to those Assets, while also registering the execution of this action.

5.3.1 Asset Management

The first action to be taken is the manual registration of new Assets. To register an Asset, the user must create the Asset JSON structure, with the necessary fields populated. Then, the user uses the register API method to register the Asset on the Ledger, that will return the same Asset structure with the ID field populated, since that information is randomly generated by the ledger itself. We can then verify the presence of said Asset in the ledger using the correct API method.

The process for the modification of an Asset is similar, with the difference of the method called, that is the modify method instead of the register one, and that the Asset the user sends already has its ID

field populated with the ID of the already existing Asset that is to be modified. It is important to note that there are fields that cannot be modified, such as the owner field. These modifications are verified by the Smart Contract and rejected.

The Asset removal method simply requires that the user supplies the Asset ID. Again, we can verify the function of this method by then querying the Ledger either for the deleted Asset, that will return an error stating the Asset was not found, or by querying the Ledger for all assets of the relevant type, and verifying that the removed Asset is not present.

Listing 5.3: Successful Asset Registration

```
1
2  wigu@DESKTOP:~$ curl --location -i \
3  > --request POST 'https://broker-module:8080/asset' \
4  > --header 'Content-Type: application/json' \
5  > --header 'Cookie: SESSION_ID=CZMZ-HyALZ1SEX4reFKTPKt084RqyZefxw%3D%3D' \
6  > --data-raw '{
7  >     "asset": {
8  >         "type": "Server",
9  >         "Location" : "Rack1",
10 >         "spec_ram_gb": 4,
11 >         "spec_cpu_cores": 8,
12 >     }
13 > }'
14
15
16 HTTP/2 200
17 content-length: 452
18 date: Sat, 25 Sep 2021 17:50:54 GMT
19 {
20     "asset": {
21         "id": "Server_i2SMtYv",
22         "type": "Server",
23         "location": "Rack1",
24         "owner": "x509::CN=user1-org1,OU=user1+OU=org1+OU=sysadmins,O=Hyperledger
25             ,ST=North Carolina,C=US::CN=org1-ca,OU=MasterThesis,
26             O=MiguelOliveira,ST=Lisbon,C=PT",
27         "spec_ram_gb": 4,
28         "spec_cpu_cores": 8,
```

```
29          "applied_tools": [],
30          "ip_addrs": [],
31          "dependencies": [],
32          "dependants": [],
33      }
34  }
```

Listing 5.3 exemplifies the Asset Registration process as done on the terminal, with lines 1 to 14 being the request, and lines 15 to 34 the response. It is important to note that all specified fields, as long as they are not mandatory, are automatically created as empty by the Ledger. We can also note that the ID was automatically generated as a form of *AssetType_ID* and that the owner field is populated with the identity of the user that submitted the request. The owner field is always compared to the identity of the user making a request in order to verify ownership and permissions related to each Asset. The Modification request is not present here due to its similarity with the Registration request, the main difference being that the Asset ID must be present and all not specified fields are not modified.

Listing 5.4: Successful Asset Deletion

```
1
2  wigu@DESKTOP:~$  curl --location -i \
3  > --request DELETE 'https://broker-module:8080/asset/Server_i2SMtYv' \
4  > --header 'Cookie: SESSION_ID=CZMZ-HyALZ1SEX4reFKTPKt084RqyZefxw%3D%3D'
5
6
7  HTTP/2 200
8  content-length: 0
9  date: Sat, 25 Sep 2021 17:54:34 GMT
```

As presented on Listing 5.4 the deletion of an Asset is carried out by sending a DELETE request with the ID of the Asset to be deleted. The system returns just an empty response with the success code (status 200 - OK), representing a successful deletion. We can then list the existing assets and verify that the deleted asset is not present (not represented here for simplicity)

5.3.2 Tool Management

The automatic Asset management, using the infrastructure management tool should be the main interaction method with the tool and the infrastructure, since it reduces the error chances when making modifications, since it automatically registers the actions made by the infrastructure management tools in the Ledger, reducing human interaction, and thus, mistake opportunities.

The normal workflow for this type of management consists of just two requests: The execution request and the confirmation request.

5.3.2.A Execution Request

The execution request is made by the user to start the tool execution process, as specified before in Section 3.2.2.C. The request, apart from the session ID, must contain the name of the tool to be used, as a request can only comprise the execution of one single plan from a single tool, and the plan files that are to be consumed by the tool, in their format and folder structure. Our tool will then analyze the dry-run of the tool and generate an Applied Tool structure with all the information that will be registered. If the tool to be used is Terraform, it will also create a list of new Assets that will be registered in the Ledger. All these generated structures are then presented to the user, along with a unique ID that represents the specific execution the user is carrying out.

The user then can see the information that represents the changes to be made to the infrastructure in the form of those data structures and confirm the execution, making a request for confirmation with the execution unique ID. In Listing 5.5 we can see an example response to the execute request made with a Terraform plan that will create two VMs in a specific Server. We can observe that the response contains the ID of the execution, an Applied Tool structure, and a list of Assets. The list of Assets represents the Assets that will be modified or created, in this case created. We must note the dependencies that are created automatically, creating a dependency relation between the host and the two VMs and that, since the tool has only run in dry-run mode, that status of the implementation is *False*. The Applied Tool structure contains all the available information about the execution of the tool, including modified or created Assets, information about the Terraform Plan files and if the tool has already implemented the changes. Since the confirmation is not yet given by the user, the changes are not implemented in the infrastructure and the final status of the execution is still empty.

Listing 5.5: Successful Asset Registration

```
1
2 wigu@DESKTOP:~$ curl --location -i \
3 > --request POST 'https://broker-module:8080/tools/Terraform/execute' \
4 > --form 'toolResources=@"terraform_plan.zip"' \
5 > --header 'Cookie: SESSION_ID=CZMZ-HyALZ1SEX4reFKTPKt084RqyZefxw%3D%3D'
6
7
8 HTTP/2 200
9 content-length: 2077
```

```
date: Sat, 25 Sep 2021 18:13:25 GMT
{
    "id": "3Dfs6fdX",
    "asset_list": [
        {
            "id": "VM_01ioKoK",
            "type": "VM",
            "location": "Server_i2SMtYv",
            "owner": "x509::CN=user1-org1,OU=user1+OU=org1+OU=sysadmins,
                      O=Hyperledger,ST=North Carolina,C=US::CN=org1-ca,
                      OU=MasterThesis,O=MiguelOliveira,ST=Lisbon,C=PT",
            "spec_ram_gb": 4,
            "spec_cpu_cores": 2,
            "applied_tools": [],
            "ip_addrs": [],
            "dependencies": [
                {
                    "dependency": "Server_i2SMtYv",
                    "origin": "Terraform_9dF8Sa2"
                }
            ],
            "dependants": [],
            "implemented": false
        },
        {
            "id": "VM_SgD5xAV",
            "type": "VM",
            "location": "Server_i2SMtYv",
            "owner": "x509::CN=user1-org1,OU=user1+OU=org1+OU=sysadmins,
                      O=Hyperledger,ST=North Carolina,C=US::CN=org1-ca,
                      OU=MasterThesis,O=MiguelOliveira,ST=Lisbon,C=PT",
            "spec_ram_gb": 4,
            "spec_cpu_cores": 2,
            "applied_tools": [],
            "ip_addrs": [],
            "dependencies": [
                {
                    "dependency": "Server_i2SMtYv",
```

```
                    "origin": "Terraform_9dF8Sa2"
                }
            ],
            "dependants": [],
            "implemented": false
        }
    ],
    "applied_tool": {
        "id": "Terraform_9dF8Sa2",
        "applied_to": [
            "Server_i2SMtYv"
        ],
        "assoc_dependencies": [
            {
                "dependency": "VM_01ioKoK",
                "origin": "Terraform_9dF8Sa2"
            },
            {
                "dependency": "VM_SgD5xAV",
                "origin": "Terraform_9dF8Sa2"
            }
        ],
        "tool_name": "Terraform",
        "file_name": "terraform_example.zip",
        "file_hash": "024addc35de4acb775bbd2ea3a59aef0852
                      fde90640ffacb234cd0177a4ba310",
        "finished": false,
        "final_state": "",
        "reverted": ""
    }
}
```


After reviewing the output of the execution request, the user has the option to either confirm the execution plan, making the tool invoke the Terraform tool to implement the changes or to discard this plan by not confirming it. In a normal workflow, the user will then confirm the plan, making the confirm request with the unique ID of the execution. This action would then trigger the execution of the tool and, after that, the user will receive an output similar to the one presented in Listing 5.5, but this time with all Assets marked as implemented and the Applied tool would then have the *final_state* field populated with the output of the tool.

We can then conclude that for this scenario, representing a realistic workflow for the usage of this tool, our tool can satisfy the requirements, by not only ensuring the execution of the plans, but also guaranteeing that all the changes are correctly registered in the Ledger, with all the information needed to identify the user responsible for the action, timestamp of the execution of each request, and without the possibility for anyone to modify, delete or tamper with these registries.

5.4 Scenario 3 - Authorization

Our solution aims to introduce both accountability and traceability to the management of IT infrastructures, and that is automatically ensured by the ledger together with the logic present on the Smart Contract, as seen in the scenario presented before. However, it is also important to ensure access control on both modifying the infrastructure or simply read read data from the inventory.

As stated on the requirements, an access control scheme is necessary. Using the Smart Contract capabilities, we implemented a simple role based access control scheme, with two different permission levels, user and admin. An user can only see and modify its Assets. When listing all Assets by type, only the ones where it is the owner will appear, and when requesting information about a specific Asset, via its ID, the tool will only return the information if the user is the owner, returning a 403-Forbidden status code if not. On the other hand, an admin can see all the Assets registered in the infrastructure and is able to modify all of them. Figure 5.1 systematizes the implemented permissions scheme in a simple diagram with two users, an admin and three assets belonging to each of the participants.

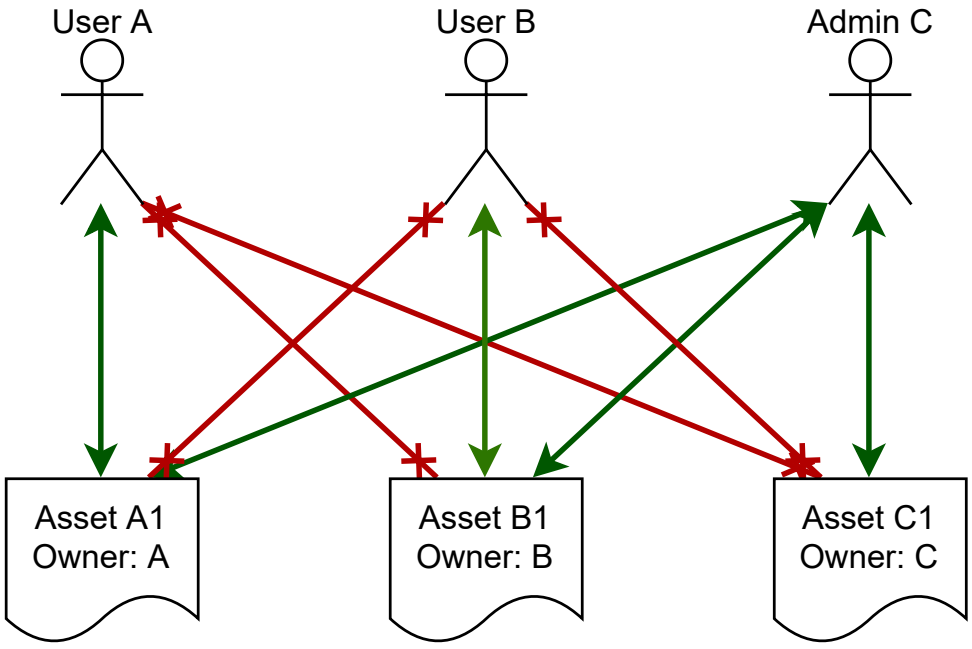


Figure 5.1: User and Admin Permissions Diagram

To test and present this behaviour, we chose to create three different users, *A*, *B* and *C*, with both *A* and *B* being normal users, and *C* being an administrator. All the users, *A*, *B* and *C*, created one Asset each, that we will just call *A1*, *B1* and *C1*, respectively. We then tested the authorization system by verifying that both *A* and *B* could only see and modify the assets of which they are the owners, and that *C* could see and modify all of the Assets. On Listing 5.6 we can observe the result of user *A* trying to query the ledger for Asset *C1*. As expected, the system returned an empty response with the status code of 403 - Forbidden, since the owner of *C1* is user *C*.

Listing 5.6: Refused Request due to lack of permissions

```
1 wigu@DESKTOP:~$ curl  -i --location \
2 > --request GET 'https://broker-module:8080/asset/VM_2fD7Se9'\
3 > --form 'identityFile=@"user_A.zip"'
4
5 HTTP/2 403
6 content-type: text/plain; charset=utf-8
7 content-length: 25
8 date: Sat, 26 Sep 2021 10:24:37 GMT
9 Insufficient permissions
```

With this scenario's verifications we can ensure not only that a simple access control scheme is possible to be implemented successfully using Smart Contracts, but also that, since the Smart Contracts are written in a general purpose programming language, it is possible to write any type of attributes into a user's identity certificate and an Asset can have any size and information in it, it is possible to implement complex access control schemes, with and indefinite group or attribute count and very high granularity. One example of a possible to implement system is the system used by Linux to specify file and directory permissions, with separate permissions for the owner, a group and others.

5.5 Scenario 4 - Dependency processing

As explained before, our solution supports the specification of dependencies between different Assets as an example of real world business logic verification. In a real IT Infrastructure, dependencies are very common, either between physical Assets (for example a Server depending on a Smart Power Distribution Unit), physical and logical Assets (for example a VM depending on its host Server) or between different virtual Assets (for example when a web server VM or Service (that can be configured as an Asset) and a backend database provider VM or Service. By programming the verification of dependencies in the Smart Contract, every time some user makes modifications to the infrastructures, the dependencies are

automatically validated, preventing the user from making changes that could break dependencies.

Dependencies may be registered into the Ledger either automatically or manually. Automatic dependency registry happens when the user uses our tool to invoke the infrastructure management tools. As demonstrated in Section 5.3, our tool automatically detects a dependency between the newly provisioned VMs, using Terraform, and their host. Since not all dependencies can be detected automatically, it is also possible for the user to both register and remove dependencies between already registered assets using the tool API. Since automatic dependency detection and registry was already proven in Section 5.3, we will now test the manual dependency registration and removal, together with testing their enforcement. For this, we create two Assets, *A* and *B*, then we register a dependency from *A* to *B* (e.g., Asset *A* depends on Asset **B**), so, as per the requirements, Asset *B* cannot be removed either before Asset *A* is removed or the dependency is removed. When trying to remove Asset *B* we observe that the Ledger denies the operation as expected, as seen on Listing 5.7. After the removal of the dependency the Ledger now allows for the deletion of Asset *B*. We also are allowed to remove Asset *B* after removing Asset *A*, since the operation of removing *A* automatically removes the dependency.

Listing 5.7: Refused Asset removal due to Dependency check

```
1  wigu@DESKTOP:~$ curl  -i --location \
2  > --request DELETE 'https://broker-module:8080/asset/VM_Erd2Kv4'\
3  > --form 'identityFile=@"user_A.zip"'
4
5  HTTP/2 403
6  content-type: text/plain; charset=utf-8
7  content-length: 40
8  date: Sat, 26 Sep 2021 11:32:41 GMT
9
10 Deletion denied due to dependency break
```

It is important, however, to note that, although both the manual and automatic dependency management proves that dependency verification as part of the business constraints' verification is a viable use case for a DL using its Smart Contracts, as shown on this scenario, automatic dependency detection is very hard to fully implement.

5.5.1 Automatic Dependency Detection Shortfall

Although the Tools Modules are developed for a single tool, enabling for very accurate parsing of both the tool plans and output, which enables the detection of some types of dependencies, such as Host-VM dependencies when using a Provisioning Tool by, for example, parsing the Terraform execution output to

check the target of the modifications and the newly provisioned resources, there are dependencies that come from the specific usage of the Assets. This type of dependencies is very difficult to detect due to the near infinite configuration possibilities for a given resource, and the fact that proof of these dependencies is usually stated either in deployed code, or in configuration files. Both these usual locations are unfeasible to be parsed due to the very large amount of very different configuration and code possibilities. A good example of this difficulty is a typical Web server dependency on a Database: evidence of this dependency will only be present in specific Web Server configuration files, or even in its code. It is unfeasible to develop a tool module that would be able to scan for all those very specific scenarios in order to detect all the dependencies. In these cases, our tool must rely on the user to manually register/unregister those dependencies on the Ledger when invoking the tools that may generate them in the real infrastructure.

5.6 Scenario 5 - Rollback of tool applied actions

This scenario is dedicated to demonstrate the rollback of actions that were made using one of the infrastructure management tools, using the process further explained in Section 5.3.2. The original goal was to develop an API endpoint that could receive the ID of the applied tool, and, with access to all the original plans from the tool, generate a new plan that would undo the original plan. However, after research both in academic publications and industry oriented support websites, it was concluded that this process is not always possible at all, and unfeasible for the majority of the remaining cases.

5.6.1 Tool Rollback Limitations

The main constraint that renders automatic rollback of infrastructure management tools actions unfeasible is the the fact that the large majority of them, including the two present in our solution, Ansible and Terraform, work on the principle that the target state is the one that must be preserved and attained. This makes sense since the objective of these tools is to read the plan, with either a declarative list of changes to apply or a target state to attain, so they usually read the current state of the resources they are to modify, and calculate a list of actions to make. The tools then apply this list of actions, bringing the state of the resources to the one desired by the user. In this process they can detect changes that are redundant or already in place, and save time and resources by not applying them twice. This also ensures idempotency, that is, if we apply the same plan more that one time, the end result should be the same as applying it once, assuming no failures occur.

However, although the tools can present the user with a list of modified parameters, they do not systematically track and store the previous state of the resource. A good example for this case is the overwriting of a file. The tool can detect if the file contents are already equal to the ones the user wants

to copy, and in that case ignore the copying operation. In this case the tool will present a ”not modified” status for that operation. However, if the file is in fact different, the tool will overwrite the file with the new version as commanded, and show a ”modified” status for the operation. The problem is that, although the tool reports that the file has been modified, it does not store the previous contents of the file.

When trying to revert an action, if the state of an operation is ”not modified”, the rollback action is to not do anything. However, if the stored state of the original operation is ”modified”, we have no way of knowing what was exactly the state of the resource before the tool execution. Continuing on the previous example of the file modification, if the state was ”not modified” the rollback action would be to do nothing. However, if the state was ”modified”, we would have no means to restore the original contents of the file, or even to know if the file existed at all originally.

The solution to circumvent this limitation would then be to modify the infrastructure management tools to make them store copies, or reports of the previous state of all modified resources, and even then there would be unpredictable possibilities of failures when reverting actions due to automatic actions taken by the systems themselves, that the tool could not account for. Other solution would be for our tool to modify all the plans that the users submitted in order to pair each action with some action that would save the state of the resource to be modified. However, this approach would be unfeasible due to the variety of modifications a tool like Ansible, for example, can make, and would suffer from the same setbacks as the previous approach of modifying the tools themselves.

However, the industry has already solved this problem with another set of tools and features, that we will present in the next section.

5.6.2 Proven Solution

Since the rollback of actions is a common event in a typical infrastructure, some solutions were devised. The main solution, since the majority of resources are in fact VMs running in some sort of hypervisor, it to then harness the features of said hypervisor to make the rollback of the state of the VMs using snapshots. The large majority of hypervisors support this technology that is the storage of the state of the VM, that can include the associated persistent memory (for example the Hard Drive, that may be virtual or not), the state of the volatile memory and even the processor state, in a persistent manner. The hypervisor is able to store all this information in a compact manner, that enables the snapshot of a resource to only occupy a fraction of the size the resource itself occupies, enabling for regular taking of snapshots that are stored for a predefined amount of time. When it is needed to rollback the state of some resource, the hypervisor is then able to get the information contained in the snapshot and setup the resource to be exactly as it was when the snapshot was taken, most of the times with unnoticeable downtime.

Another solution that can be used is to resort to some file systems’ ability to create snapshots of

the entire file system, or to maintain a log of all the modifications, enabling the user to revert the file system state to a previous state. This method is usually employed in physical resources, and has some limitations due to the fact that only the file system's state is restored, and volatile memory contents are lost. Usually this method involves a reboot of the resource so that the OS reloads all information from the file system.

5.6.3 Integration with our solution

The integration of automatic rollback operations in our tool is possible, however, it would require for the hypervisor to be connected to our solution as a Infrastructure management tool, a situation that we consider as being out of scope for this work.

Nonetheless, the rollback of the state of Assets is fully supported by our tool in a manual manner. The process for rollback then consists of the user triggering said rollback manually in the correct tool/hypervisor, and then registering it in our tool, so that the inventory database continues to represent the actual state of the real infrastructure. The user accessible API implements methods that allow the user to both delete Assets, modify them, to mark Applied Tools as reverted, or to even delete them. With the set of methods, the user can, manually, undo the changes made after the moment to which the real resource was rolled back to. For example, if the user is to rollback the execution of an Ansible playbook, it would then do that rollback manually in the infrastructure, for example using the snapshot feature of the hypervisor hosting the target of said playbook, and then deleting the Applied Tool registered in the Ledger, which would, in turn delete all possible dependencies created by that Applied Tool, effectively rolling back the state of the Ledger too.

5.7 Load and throughput

After the qualitative analysis presented in the different scenarios above, we will also make an evaluation on the performance of this tool. Since the tool is intended to be used as a middle layer between the user and the infrastructure or the infrastructure management tools, it is important to ensure that the tool is then able to serve multiple requests in parallel, while also adding a minimal time delay to the operations that are relayed through this tool. We will then test both the throughput and response times of different requests made to the tool, testing both the variability between different request types, mainly read and write, and if there is any noticeable performance drop when the number of Assets registered in the State database increases.

For all the tests, we used Apache JMeter[3] to generate the requests and collect results. JMeter allows us to create any type of request, set the concurrency level (number of concurrent requests) and

[3] https://jmeter.apache.org/, accessed 27th September 2021

the total number of requests to be executed. It then collects information about every request and its response. For our tests, the main metrics we collected were the response time, in milliseconds, the general throughput (the number of requests answered per second) and lastly, as a control the success rate based on the HTTP return status of every request.

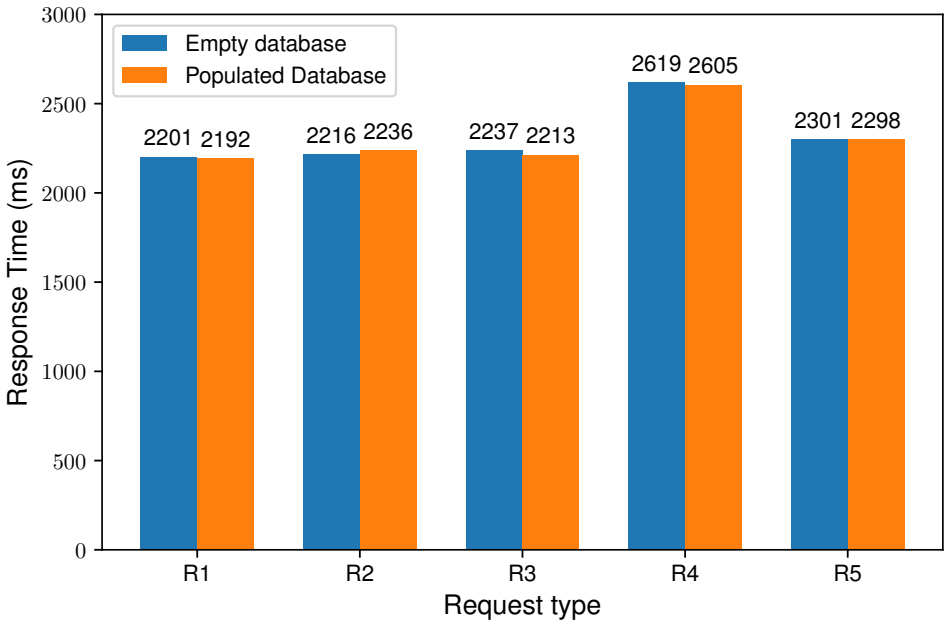
Since our solution has several API endpoints, we will make throughput evaluations only for the most relevant ones:

- Read Requests:
  - R1 - Get Asset by ID
  - R2 - Get Applied Tool by ID
- Write Requests:
  - R3 - Register Asset
  - R4 - Modify Asset
  - R5 - Register Applied Tool

Both the login and Asset Type handlers were excluded from this evaluation since they are less used endpoints, and where performance is not as important. However, these requests, being very simple from a Ledger perspective, are in fact faster than most other requests. The request to get assets by their type is also not evaluated since its response time is directly dependent from the number of assets with the specific type the user has access to, with the bottleneck in this request being the transmission of possibly very large amounts of data due to a large number of Assets. It is also important to note that the automatic tool execution endpoints are also not benchmarked since they depend heavily on the execution time of the tools, that is out of scope for this project, and all the requests internally made to the ledger are the same as the manual requests that we are already evaluating.

We will benchmark each request in two different scenarios: with a minimal number of Assets registered in the ledger, and then with the ledger populated with around 100000 Assets, in order to simulate a large infrastructure. We can then compare the response times and throughput in both scenarios and verify if there is any performance degradation with the increased number of stored assets. For each scenario, we will run each request 1000 times with a parallelism of 100 concurrent requests, and average the response times and throughput, in order to get a significant performance measure. The number of 1000 executions was chosen due to it being a large enough value that allows for the tool to stabilize, and provide consistent results. The number of 100 concurrent requests was obtained by testing different values of concurrency until the throughput value maximized and stabilized, due to the tool being working at 100% capacity. With larger concurrency values the throughput maintained, however increasing response times. With lower values the response times did not change, however making the throughput

value decrease. However, it is important to note that in experiments with larger core counts and larger RAM sizes, the throughput increased with increased concurrency values. The standard deviation values for the response times in each test were also recorded, being always lower than 100ms, meaning that there is little dispersion of values between all the individual requests in each test.

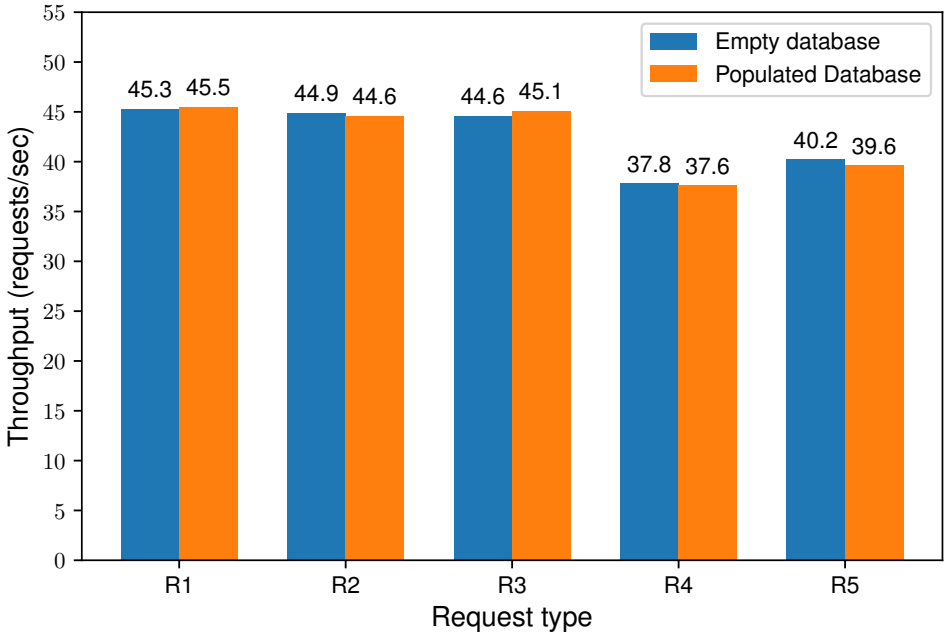


R1 - Get Asset by ID; **R2** - Get Applied Tool by ID; **R3** - Register Asset; **R4** - Modify Asset; **R5** - Register Applied Tool;

Figure 5.2: Response times for different request types

As we can observe from both graphics in Figure 5.2 and Figure 5.3, as expected, there is a inverse correlation between response times and throughput: when the response time is greater, the throughput is lower. However, when repeating the tests with increased computing resources allocated to the Ledger, by increasing the number of concurrent requests, the throughput values increased substantially (when tested with 6 CPU cores and 1000 concurrent requests, the tool achieved throughputs of around 200 requests per second). This is one factor that indicates that the requests are processed independently of each other by the ledger, with the limit being the concurrent processing capacity of the ledger nodes. This results are on par with the ones published in several studies about performance of distributed Ledgers, and more specifically Hyperledger Fabric such as [22, 24].

We can also observe in the graphics that, unexpectedly, both read and write methods achieve similar performances, both in response time and throughput. All the methods achieved similar metrics, with the exception of the Asset modification method. However, the lower performance of the Asset modification method can be explained because, for the Smart Contract to approve a modification of some Asset, it must first read it from the state database in order to make all the necessary verifications for the validity of all modifications before writing the updated Asset version to the Ledger.



R1 - Get Asset by ID; **R2** - Get Applied Tool by ID; **R3** - Register Asset; **R4** - Modify Asset; **R5** - Register Applied Tool;

Figure 5.3: Throughput for different request types

Also unexpectedly, methods with different complexity levels of logic programmed into the Smart Contract behaved similarly. However, this behaviour can also be explained easily, by understanding that the main time taking operations in a request are not part of the Smart Contract execution, but in all the computations that the Ledger must execute in order to process a transaction, such as the process to launch a new process to run the smart contract instance dedicated to each transaction, the cryptographic signing and verification operations, and the ordering of nodes by the ordering service, that involves reaching consensus between all orderers. The network delays may also introduce some latency in the requests, although in our testing said delay is minimized since all nodes of the Ledger run in containers in the same virtual machine. We can then also conclude that the execution of the Smart Contracts in very fast, allowing for more complex logic to be programmed in them without major performance concerns.

Another conclusion that we can draw from the obtained results is that there is no noticeable performance degradation from the increase of number of assets in the database and nodes in the ledger blockchain itself. A value of around 100000 assets in the database has been chosen as the target for a populated database. We chose this value since we consider that, in the infrastructures this tool may be used, it is very improbable to achieve such a high number of assets. Even in some requests, such as request *R1* - Get an Asset by its ID, the throughput increased slightly, and the response time decreased, when the database was populated. Although these differences are well within a margin of error, this demonstrates that the performance degradation, if existing, is negligible when compared to the observed values (for example a difference of 9ms in response time in a gross value of around 2200), and

we can extrapolate that this trend is expected to keep itself for larger numbers of assets in the database. This proves that the Hyperledger Fabric database implementation, LevelDB, has a well formulated data storage algorithm that is performance resilient against large amounts of stored data.

Although the response times for the requests are noticeable, since they are around 2-3 seconds per request, the capacity of the system to process a large number of requests in parallel allows for throughput figures that are much higher that would be possible without parallelism. With serial execution of requests, having in mind an average response time of 2200ms per request, the throughput would be as low as 0.45 requests per second. Having in mind the purpose of this tool, and that all the requests that are processed by it are related to IT Infrastructure's management activities, the throughput of the tool will then be more important that the response time, since many of the infrastructure management activities, such as provisioning a resource or implementing changes take considerably longer that the response times of the tool requests, making them not very noticeable in the context of the activities. With this in mind, and focusing of the throughput values, we observed that with, with somewhat limited resources (since all the nodes of the Ledger are running in the same virtual machine, that itself has just 6 CPU cores and 4GB of RAM), the throughput established itself above 35 requests per second for all request types. Testing with larger resource availability (either increasing the specifications of the Ledger VM or increasing the number of simultaneous requests) showed increased performance, that is in par with the general Ledger performance evaluations presented in other studies such as [22, 24]. From a performance perspective, this makes us confident that this tool can easily cope with large numbers of infrastructure modifications per seconds, as it can happen in the real world.

5.8 Result Discussion

After the presentation of both qualitative and quantitative evaluations of the developed solution, we are now able to make a complete evaluation of the tool as a whole, taking into account the different conclusions achieved in each of the test scenarios and the performance evaluation.

From a qualitative standpoint, the presented scenarios ought to cover most of the common real life operations, while providing results that show that the DL system has potential for the expansion of requirements, allowing for more complex business logic, embedded in the Smart Contracts. However, while we could fully implement and satisfy the scenarios for Login and session management, authorization and registered dependency processing, it must be noted that the automatic mechanisms for analysis of changes made by infrastructure management tools present a greater challenge that previously anticipated. While this was not evident in scenario 2 - Normal Workflow, due to the usage of Terraform and only host-VM dependencies, it was further investigated in the scenarios of Dependency Processing and Tool Rollback. While the processing of already registered dependencies is totally supported and

can be implemented in Smart Contracts to evaluate more complex dependencies, their detection is a very complex problem, as it is detailedly discussed in Section 5.5.1, and has consequences that impact the feasibility of tool rollback without the usage of snapshotting or similar technologies, as presented in Section 5.6.

From a performance standpoint, we analyse both throughput and response times for this solution. The great parallelization capabilities of both the Ledger and the developed modular system to interface with it made this tool able to be scaled up in performance, allowing for it to satisfy even large infrastructure requirement. Furthermore, even in our baseline testing, with very limited resources, the tool presented good throughput performance, processing more than 35 requests per second across all types of requests, number that we consider above the needs for most small infrastructures, with up to 100 devices, with limited resources. For larger infrastructures the tool can scale up to allow for larger throughputs. On the response time side, the ledger system introduces a response time for, at least, 2200ms per request. However, this value is constant for all types of requests since it is related with the processing of requests by the Ledger itself, and not due to the complexity of the code in the smart contracts, since it presents very little variation between requests with very different processing requirements for the execution of the related Smart Contract function. Because of this, we can expect that the response times are not very dependent on the complexity of the business logic, allowing for more complex business logic validations with minimal performance penalties. Furthermore, we consider that, in the context of the managements activities, these response times are not too significant.

6

Conclusion

Contents

6.1 Objectives 69

6.2 Conclusions 70

6.3 System Limitations and Future Work 71

After the development of the solution and its evaluation, we can now draw conclusions about the original objectives of this work, its requirements, and how they were achieved, and if, in out particular case, there is viability for the usage of a DL as a central component in the logging and access control of IT infrastructures management actions in a secure way, providing both access control and traceability of past actions.

6.1 Objectives

In this Masters Thesis, we proposed to verify the viability of harnessing the capabilities of a DL to enable a secure and traceable way of managing IT Infrastructures. For this, we proposed to develop a proof-of concept solution that would include not only the coding of Smart Contracts in the Ledger to express the business logic, but also to create an interface both for the users to connect to, but also to integrate the execution of IT Infrastructure Management tools and the registering of the modifications in the Ledger. We started our work by researching about the existence of similar solutions, having found that although there exist many IT Infrastructure Management tools, they do not provide significant means of creating an inventory of the entire infrastructure, and don't offer any security related features. On the other side, there are several DL solutions, although none dedicated to the management of IT Infrastructures. However, many of the DL solutions are modular and configurable to attend to different needs and specifications.

We chose to use Hyperledger Fabric for our solution, since it is one of the best performing DL solutions, while offering a very large degree of configurability, and large support and extensive documentation. We also chose two Infrastructure Management tools, Ansible and Terraform, to demonstrate the possible integration of a solution like ours with already existing tools, improving security and reducing the space for mistakes. Ansible and Terraform were chosen due to their significance in the industry, being the most used tools to Configure and Deploy resources, respectively. They support a large number of devices and endpoints, making them very versatile.

A main goal for this solution was for it to be able to register every modification to the infrastructure in an immutable way, and to register the authors of every action, thus enabling the creation of both an inventory of the infrastructure, but also a log of changes that is immutable and that can trace back all modifications to its author. Hyperledger Fabric presents the solution for this, by not only supporting the identity creation and management, needed to create identities, and credentials, for all users, but by automatically having a state database, that can store the inventory of the infrastructure, and stores the modifications in a blockchain, that is an immutable storage structure by design. Then we could use the Smart Contract capabilities of Fabric to implement access control based on the attributes that we can embed in the user's identity. This way we could develop an Access Control mechanism, not only for

writing information to the ledger (and state database), but also for reading it, based on the calling user's attributes. As explained in Section 5.4, this function can be expanded to support much more complex Access Control schemes.

Hyperledger Fabric's Smart Contract capabilities were explored to implement not only authentication and Access Control, but also Business Logic. Since there are many rules that may be specified to filter out the possibility of executing certain management tasks, we chose to demonstrate the ledger capabilities by implementing the concept of dependencies between assets, making the Ledger, through the logic in the Smart Contracts, be able to automatically evaluate each and every task that the user submits, either by manual submission or as part of a IT infrastructure management tool execution, and approve or deny it based on the creation and modification of assets and their dependencies, in a way that no dependency is broken by making changes.

A modular tool was then developed to support both the users connections but also to integrate the tools, Ansible and Terraform, with the Ledger. The system is developed with a microservices architecture, where each module has a specific purpose and is as generalized as possible to enable modifications in one module without the need for modifying all other modules. This enables for new tools to be added and removed as needed. The tool integration was developed as a proof-of-concept, only allowing for the execution of already formulated plans, and basic detection of tool-induced changes, that are then registered in the Ledger.

6.2 Conclusions

After the development of the solution, evaluation was carried, in a qualitative and quantitative perspectives. The solution was evaluated in a qualitative manner for the compliance with the requisites previously defined in Section 3.1, by making use of several evaluation scenarios that aimed not only to cover the requisites but also to represent typical workflows in which the tool is part of. The access control mechanism was tested for both the denial of access to unauthenticated users, but also for the denial of access to unauthorized users, and the tool behaved as expected, denying all requests that were not valid according to the Access Control rules. Scenarios for both manual manipulation of assets and dependencies were also proposed, with the tool being able to register and delete assets, according to the business logic rules, and to manipulate dependencies, and deny requests that would have broken dependencies. Scenarios for the testing of workflows involving tool execution and automatic asset tracking and dependency creation were also evaluated, with the tool being able to detect new assets and dependencies, although with limitations.

From a quantitative perspective, a performance analysis was performed, and we could conclude that, although the tool introduces a latency penalty in each request and modification made to the infrastruc-

ture, by presenting response times of around 2200ms for all requests, it can handle a large number of concurrent requests, presenting a throughput of more than 35 requests per second with limited Ledger resources. This value can be vastly increased by increasing the resources available to the solution, as explained on Section 5.7.

From the obtained results, we could then conclude that the usage of a DL, more specifically Hyperledger Fabric, can be a good solution to ensure that all actions that may modify an IT Infrastructure are both filtered and processed, to ensure the verification of compliance with Infrastructure constraints, such as dependency checking, the verification of the permissions of a user to do such modifications, and lastly to ensure that all the modifications are registered in an immutable way, that enables accountability and traceability for all actions. Additionally, Fabric deploys a state database, that is closely tied with the ledger, that can represent the infrastructure, acting as its inventory, without the need for the implementation of external databases and the consequent development of mechanisms to ensure that all information that is written to the database is also written in the ledger.

6.3 System Limitations and Future Work

As previously stated, the developed solution is a proof-of-work, and, consequently, may be object of further investigation and development. All the logic that is present in the Smart Contract can be expanded both to enable for the verification of more and more complex constraints and finer grained access control, although we could already conclude that from a feasibility perspective, such expansions are possible. The biggest limitations of this solution come from two main components: the Ledger and the Tools. As stated in Section 5.7, the Ledger introduces some delay in all requests, making the user having to wait some time for each request to complete. Optimizations in this aspect can be object of further study, by trying to reduce response times for example for read requests where the full mechanism for transaction evaluation may not be needed. From the tools perspective, and as explained in detail in Section 5.6, the detection of dependencies in tool output is very limited in the present version, and is a complex problem due to the variety of sources for dependencies. Further study in this area could be useful to enable solutions like the one presented in this Thesis to have a more complete inventory of the infrastructure without the need for humans to manually register details of the infrastructure, instead having the system automatically detect more of those details.

Bibliography

[1] M. Hilton, "Understanding and Improving Continuous Integration," in *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ser. FSE 2016. New York, NY, USA: Association for Computing Machinery, 2016, pp. 1066–1067.

[2] J. Itkonen, R. Udd, C. Lassenius, and T. Lehtonen, "Perceived Benefits of Adopting Continuous Delivery Practices," in *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, ser. ESEM '16. New York, NY, USA: Association for Computing Machinery, 2016.

[3] M. Oliveira and R. S. Cruz, "Ensuring Traceability on Management of IT Infrastructures : Orchestrator based on a Distributed Ledger," in *2021 16th Iberian Conference on Information Systems and Technologies (CISTI)*, 2021, pp. 1–5.

[4] U. Pawar and M. Bhelotkar, "Virtualization: A Way towards Dynamic IT," in *Proceedings of the International Conference & Workshop on Emerging Trends in Technology*, ser. ICWET '11. New York, NY, USA: Association for Computing Machinery, 2011, pp. 262–263.

[5] R. Dua, R. Raja, and D. Kakadia, "Virtualization vs Containerization to Support PaaS," in *2014 IEEE International Conference on Cloud Engineering*, 2014, pp. 610–614.

[6] J. Watada, A. Roy, R. Kadikar, H. Pham, and B. Xu, "Emerging Trends, Techniques and Open Issues of Containerization: A Review," *IEEE Access*, vol. 7, pp. 152 443–152 472, 2019.

[7] C. Pahl, "Containerization and the PaaS Cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.

[8] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "OpenFlow: Enabling Innovation in Campus Networks," *SIGCOMM Comput. Commun. Rev.*, vol. 38, no. 2, pp. 69–74, 2008.

[9] P. Selvaraj and V. Nagarajan, "Migration from conventional networking to software defined networking," in *2017 International Conference on IoT and Application (ICIOT)*, 2017, pp. 1–7.

[10] A. Vahdat, D. Clark, and J. Rexford, “A Purpose-Built Global Network: Google’s Move to SDN: A Discussion with Amin Vahdat, David Clark, and Jennifer Rexford,” *Queue*, vol. 13, no. 8, pp. 100–125, 2015.

[11] Google.com. (2020) Google Cloud Computing, Hosting, Services & APIs. Google. Accessed 10-September-2021. [Online]. Available: https://cloud.google.com/gcp

[12] Amazon.com. (2011) Amazon Web Services (AWS) - Cloud Computing Services. Amazon Web Services, Inc. Accessed 10-September-2021. [Online]. Available: https://aws.amazon.com/

[13] E. Androulaki, A. Barger, V. Bortnikov, C. Cachin, K. Christidis, A. De Caro, D. Enyeart, C. Ferris, G. Laventman, Y. Manevich, S. Muralidharan, C. Murthy, B. Nguyen, M. Sethi, G. Singh, K. Smith, A. Sorniotti, C. Stathakopoulou, M. Vukolić, S. W. Cocco, and J. Yellick, “Hyperledger Fabric: A Distributed Operating System for Permissioned Blockchains,” in *Proceedings of the Thirteenth EuroSys Conference*, ser. EuroSys ’18. New York, NY, USA: Association for Computing Machinery, 2018.

[14] Cisco.com. (2020) Software-Defined Networking (SDN) Definition - Cisco. Cisco Systems, Inc. Accessed 25-September-2021. [Online]. Available: https://www.cisco.com/c/en/us/solutions/software-defined-networking/overview.html#~benefits

[15] Juniper.net. (2020) SDN, Network Management, and Operations — Juniper Networks. Juniper Networks, Inc. Accessed 6-October-2021. [Online]. Available: https://www.juniper.net/us/en/products-services/management-operations-sdn/

[16] HPE.com. (2020) OneView IT Infrastructure Management Software. Hewlett Packard Enterprise. Accessed 20-September-2021. [Online]. Available: https://www.hpe.com/us/en/integrated-systems/software.html

[17] DellTechnologies.com. (2020) Dell EMC Managed Services. Dell Technologies. Accessed 20-September-2021. [Online]. Available: https://www.delltechnologies.com/en-us/services/infrastructure-managed-services.htm

[18] NuageNetworks.com. (2020) (SDN) Telco Cloud - Nuage Networks. Nuage Networks. Accessed 5-October-2021. [Online]. Available: https://www.nuagenetworks.net/solutions/telco-cloud/

[19] Ansible.com. (2020) Ansible Tower — Ansible.com. Red Hat / Ansible. Accessed 19-September-2021. [Online]. Available: https://www.ansible.com/products/tower

[20] M. Schinle, C. Erler, P. Andris, and W. Stork, “Integration, Execution and Monitoring of Business Processes with Chaincode,” in *2020 2nd Conference on Blockchain Research Applications for Innovative Networks and Services (BRAINS)*, 2020, pp. 63–70.

[21] F. Dai, Y. Shi, N. Meng, L. Wei, and Z. Ye, “From Bitcoin to cybersecurity: A comparative study of blockchain application and security issues,” in *2017 4th International Conference on Systems and Informatics (ICSAI)*, nov 2017, pp. 975–979.

[22] N. Kannengießer, S. Lins, T. Dehling, and A. Sunyaev, “Trade-Offs between Distributed Ledger Technology Characteristics,” *ACM Comput. Surv.*, vol. 53, no. 2, 2020.

[23] W. Cai, Z. Wang, J. Ernst, Z. Hong, C. Feng, and V. Leung, “Decentralized Applications: The Blockchain-Empowered Software System,” *IEEE Access*, vol. 6, pp. 53 019–53 033, 2018.

[24] R. Nadir, “Comparative study of permissioned blockchain solutions for enterprises,” in *2019 International Conference on Innovative Computing (ICIC)*, nov 2019, pp. 1–6.

[25] S. Nakamoto, “Bitcoin: A peer-to-peer electronic cash system,” *Decentralized Business Review*, p. 21260, 2008.

[26] D. Eastlake and P. Jones, “Rfc3174: Us secure hash algorithm 1 (sha1),” Network Working Group, USA, Tech. Rep., 2001.

[27] S. Bhalerao, S. Agarwal, S. Borkar, S. Anekar, N. Kulkarni, and S. Bhagwat, “Supply Chain Management using Blockchain,” in *2019 International Conference on Intelligent Sustainable Systems (ICISS)*, 2019, pp. 456–459.

[28] M. Hulea, O. Rosu, R. Miron, and A. Aştilean, “Pharmaceutical cold chain management: Platform based on a distributed ledger,” in *2018 IEEE International Conference on Automation, Quality and Testing, Robotics (AQTR)*, 2018, pp. 1–6.

[29] S. Rouhani, R. Belchior, R. Cruz, and R. Deters, “Distributed Attribute-Based Access Control System Using a Permissioned Blockchain,” 2020.

[30] G. Nyame, Z. Qin, K. Agyekum, and E. Sifah, “An ECDSA approach to access control in knowledge management systems using blockchain,” *Information (Switzerland)*, vol. 11, no. 2, p. 111, feb 2020.

[31] K. Rani and C. Sharma, “Tampering Detection of Distributed Databases using Blockchain Technology,” in *2019 Twelfth International Conference on Contemporary Computing (IC3)*, 2019, pp. 1–4.

A

Sample Smart Contract Code

In this appendix we present sample Smart Contract Functions for Registering, Removing and Getting Assets, to exemplify how constraints can be coded and specified, more specifically the access control and dependency checking, in the context of a Smart Contract. Must be noted that some error handling functions are not depicted for clarity.

Listing A.1: Asset Registration

```
1 func (s *SmartContract) RegisterAsset(ctx contractapi.TransactionContextInterface
      ↪ , asset *AssetRepresentation.Asset) (*AssetRepresentation.Asset, error) {
2     clientID, err := s.GetSubmittingClientIdentity(ctx)
3
4     // validate type
5     trackerJson, err := ctx.GetStub().GetState("TypeTracker")
6     if trackerJson == nil {
7         return nil, fmt.Errorf(" Type Tracker not found")
8     }
```

```
9
10      tracker := AssetRepresentation.TypeTracker{}
11
12      err = json.Unmarshal(trackerJson, &tracker)
13
14      if !s.Contains(tracker.AssetTypes, asset.Type) {
15          return nil, fmt.Errorf("specified asset type does not exist")
16      }
17
18      // generate random asset id
19      newRandID := s.RandStringBytes(7)
20
21      newID, err := ctx.GetStub().CreateCompositeKey(asset.Type, []string{newRandID
            ↪ })
22      newIDReadable, err := s.CompToReadableKey(ctx, newID)
23
24      // populate remaining fields
25      asset.ID = newIDReadable
26      asset.Owner = clientID
27      asset.Implemented = false
28
29      // doesn't make sense to create an asset with dependants
30      asset.Dependants = make([]*AssetRepresentation.DependantRelation, 0)
31
32      // populate if empty
33      if asset.IpAddrs == nil || len(asset.IpAddrs) == 0 {
34          asset.IpAddrs = make([]string, 0)
35      }
36
37      // we have to check if any dependencies are created and implement them as
38      // dependants too for that, we get all dependency assets, and add this
39      // new asset as dependant
40      if asset.Dependencies == nil || len(asset.Dependencies) == 0 {
41          asset.Dependencies = make([]*AssetRepresentation.DependencyRelation, 0)
42      } else {
43          for _, dependencyObj := range asset.Dependencies {
44              dependencyAsset, depAssCompId, err := GetAssetIntern(s, ctx,
                    ↪ dependencyObj.Dependency)
```

```
45
46              newDependant := AssetRepresentation.DependantRelation{
47                  Dependant: newIDReadable,
48                  OriginID:  "Dependant_Creation",
49              }
50
51              dependencyAsset.Dependants = append(dependencyAsset.Dependants, &
                    ↪ newDependant)
52              dependencyAssetJson, err := json.Marshal(dependencyAsset)
53              err = ctx.GetStub().PutState(depAssCompId, dependencyAssetJson)
54          }
55      }
56
57      if asset.AppliedTools == nil || len(asset.AppliedTools) == 0 {
58          asset.AppliedTools = make([]string, 0)
59      } else if len(asset.AppliedTools) > 1 {
60          return nil, fmt.Errorf("too many applied tools for asset creation: %v",
                ↪ err)
61      } else {
62          _, _, err = GetAssetIntern(s, ctx, asset.AppliedTools[0])
63          if err != nil {
64              return nil, fmt.Errorf("unable to find applied tool asset: %v", err)
65          }
66      }
67
68      assetJSON, err := json.Marshal(asset)
69      err = ctx.GetStub().PutState(newID,assetJSON)
70      return asset, nil
71  }
```

Listing A.2: Remove Asset

```
1  func (s *SmartContract) RemoveAsset(ctx contractapi.TransactionContextInterface,
       ↪ assetId string) error {
2      isAdmin, err := s.CheckUserAdmin(ctx)
3      if err != nil {
4          return fmt.Errorf("could not verify if user is admin: %v", err)
5      }
```

```
6
7       clientID, err := s.GetSubmittingClientIdentity(ctx)
8       if err != nil {
9           return fmt.Errorf("could not get client ID: %v", err)
10      }
11
12      asset, compositeId, err := GetAssetIntern(s, ctx, assetId)
13      if err != nil {
14          return fmt.Errorf("could not get asset: %v", err)
15      }
16
17      if !(isAdmin || clientID == asset.Owner) {
18          return fmt.Errorf("access denied to change asset")
19      }
20
21      err = ctx.GetStub().DelState(compositeId)
22      if err != nil {
23          return fmt.Errorf("could not remove asset in ledger: %v", err)
24      }
25      return nil
26  }
```

Listing A.3: Get Asset

```
1  func (s *SmartContract) GetAsset(ctx contractapi.TransactionContextInterface,
       ↪ assetId string) (*AssetRepresentation.Asset, error) {
2       isAdmin, err := s.CheckUserAdmin(ctx)
3       if err != nil {
4           return nil, fmt.Errorf("could not verify if user is admin: %v", err)
5       }
6
7       clientID, err := s.GetSubmittingClientIdentity(ctx)
8       if err != nil {
9           return nil, fmt.Errorf("could not get client ID: %v", err)
10      }
11
12      asset, _, err := GetAssetIntern(s, ctx, assetId)
13      if err != nil {
```

```
14          return nil, fmt.Errorf("could not get asset: %v", err)
15      }
16
17      if !(isAdmin || asset.Owner == clientID) {
18          return nil, fmt.Errorf("asset with id %v not found", assetId)
19      }
20
21      return asset, nil
22  }
```