Distributed Ledger Technology to Enable Secure Management of IT Infrastructures

Development and evaluation of a Proof-of-concept tool using Hyperledger Fabric

Miguel Oliveira
Department of Computer Science and Engineering
Instituto Superior Técnico, Universidade de Lisboa
Lisbon, Portugal
moreira.oliveira@tecnico.ulisboa.pt

***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 de 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.**

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

I. Introduction

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 services providers, and in smaller ones of private businesses that still maintain their own physical infrastructures. These changes were needed 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. To enable these changes, new technologies, such as virtualization, containerization and Software Defined Networking (SDN) were employed, aiming to separate the underlying hardware from the software running in those infrastructures[3].

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.

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.

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. 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)[3].

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 commited[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[3].

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.

In the following, Section II explores the current state of the art, and related works, Section III describes the chosen architecture, the requirements for implementation and the main aspects of the implementation of the tool, Section IV provides explanations for several design choices, Section V presents the proposed methodology for the evaluation of the work, and its evaluation, and Section VI presents a summary of the whole document.

II. State of the Art

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.

A. 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. 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.

B. Distributed Ledger Technology

The DLT ensures the logging of information in a highly available, append-only database by using physically distributed storage and computing devices, even in an untrustworthy environment. There are many implementations of this technology, with different objectives and employing different designs, based on Blockchains or Directed Acyclic Graphs (DAGs)[4]. In general, DLs are tools and applications based on DLT, i.e, delivering highly available, append-only distributed databases working on untrustworthy environments, where Byzantine failures can happen, such as crashed or unreachable nodes, occurrence of significant network delays, and even malicious behavior of nodes. DLs are comprised of separate Nodes that work together to maintain a consistent state of the replicated ledger across all the nodes [4]. Some comparative studies about DLs are presented in [4], [5], and [6]. We will focus our work on the Blockchain based DLs because of the blockchain capacity for ensuring that only exists one concurrent state on the ledger, opposed to DAGs, where many concurrent states can coexist. 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[5], [7]. This structure ensures, by design, the requirement for immutability of the stored logs. Additionally, the Blockchain implementation in the DLs is a private one. This means that access to the blockchain and the information stored in it is controlled, for both reading and writing by code external to the chain, the Ledger itself. This design, opposing to the Public blockchains, does not require any algorithm to regulate the addition of new blocks such as proof-of-work, as used on many public blockchain implementations like Bitcoin or Etherium[7], relying just on consensus algorithms to decide

on the order of the blocks added to the blockchain across network nodes[8].

1) Smart Contracts: Smart contacts, also called Chaincode, are programs that are executed in order to change the state of the ledger[9]. 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[9], [10]. This software layer enables custom processing of all the actions submitted to the ledger, enabling both fine grained access control 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[9], [11], [12]. 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[8].

2) Hyperledger Fabric: Hyperledger Fabric is a "modular and extensible open-source system for deploying and operating permissioned blockchains"[8]. Since it is intended to be used as a part of bigger solutions and systems, it is very configurable, modular, and offers complete Application Programming Interfaces (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, responsible for the execution of smart contracts and maintaining the state database, and for the maintaining of the blockchain, ordering the blocks that are to be added, respectively[8].

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), although any CA can be used. 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[8].

As explained in detail in [8], 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.

3) Using Distributed Ledger Technology: As explained in [9], 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 [13] 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.

The work presented in [11] 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 [12] 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

C. 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.

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.

III. Solution Architecture

A. Approach

Our 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, are:

- **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 Virtual Machine (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;

B. General Architecture

In order to satisfy the requirements for adaptability and modularity, a general architecture for the solution was designed. As shown on Figure 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.

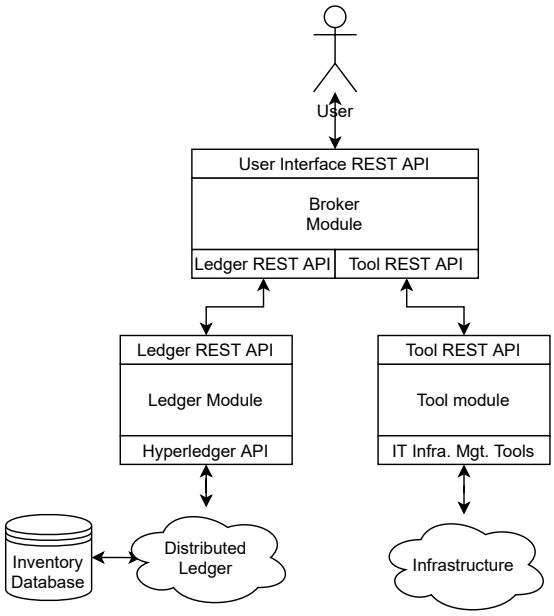


Fig. 1. General Solution Architecture

1) 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.

2) 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) 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**.

C. 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**.

1) Login: Since the tool supports **Session based** access, a Login functionality is needed. 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 makes 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.

2) 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. The user will 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.

3) 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. 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. It is important to note that all modifications to the infrastructure are always verified and written in the Ledger before being executed over the infrastructure.

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.

D. Infrastructure Management Tools

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.

1) 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.

2) 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 Operating Systems (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).

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

E. 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 II-B2, that increases the speed of the Ledger to close to 2000 Transactions per Second (TPS), a great increase when compared to similar solutions[6];
- 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.

IV. Implementation Details

A. Environment

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.

[9]https://www.hyperledger.org/, accessed on 10th September 2021

[10]https://www.linuxfoundation.org/, accessed on 10th September 2021

B. 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. The chosen topology for the Ledger system followed the documentation recommendations of having:

- **Two Organizations** - Org0 for the orderer service, and Org1 for the peers and clients (users);
- **Three CAs** - One to provide TLS certificates for communication between nodes and one for identity creation withing each organization;
- **Three orderer nodes** - All belonging to Org0;
- **Two peer nodes** - Belonging to Org1;
- **One channel** - Only one blockchain and state database that stores all information;

To provide identity management, we 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.

C. Smart Contracts

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.

The Assets have four main associated functions, to Get, Register a new Asset, Modify an existing Asset, and Remove an Asset. It is necessary to enforce 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.

From a business logic perspective, the registration of a new Asset verifies 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. For the removal of an Asset, 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. 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 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.

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 and access control verification. 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.

V. Evaluation and Result Analysis

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.

A. 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 Section IV, 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.

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. When supplied with incorrect credentials, 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 and that the process of obtaining credentials via brute force is unfeasible since it is unfeasible to generate the needed certificate signature, since the certificate must be signed by the Organization 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.

B. Scenario 2 - Normal Workflow

This scenario is dedicated to the demonstration of the capabilities of the tool for the support of a normal infrastructure management workflow, without considering failures. 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.

1) Asset Management: 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.

We could verify that all three methods worked as expected, with the system creating, modifying and deleting the Assets as requested.

2) Tool Management: The automatic Asset management should be the main interaction method with the tool and the infrastructure, since it reduces the error chances when making modifications, by automatically registering 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.

The execution request is made by the user to start the tool execution process. The request must contain the name of the tool to be used and the plan files that are to be consumed by the tool. 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. 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. After an execute request made with a Terraform plan that will create two VMs in a specific Host, we could 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. Dependencies 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. Since the confirmation is not yet given by the user, the changes are not implemented in the infrastructure and the final status of the tool execution is still empty.

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 this scenario we confirmed the plan. This action triggered the execution of the tool and, after that, we received an output similar to the one of the plan request, 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 these registries.

C. Scenario 3 - Authorization

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.

To test and present this behaviour, we chose to create three different users, one being an admin and the others being normal users. We could then verify that each user could only see and modify its Assets, while the admin could see and modify all Assets.

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.

D. Scenario 4 - Dependency processing

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. 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. To test dependency processing, two Assets were created with a dependency between them. We could then check that the tool refuses deletion of Asset the other Asset depends on before either the dependency of the depending Asset are removed.

E. 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. 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.

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. The tools read the current state of the resources they are to modify and the supplied plans or target state, 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.

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. When trying to revert an action, if the state of an resource in the report of original action 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.

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. 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.

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.

F. 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.

For all the tests, we used Apache JMeter[11] to generate the requests and collect results. 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:

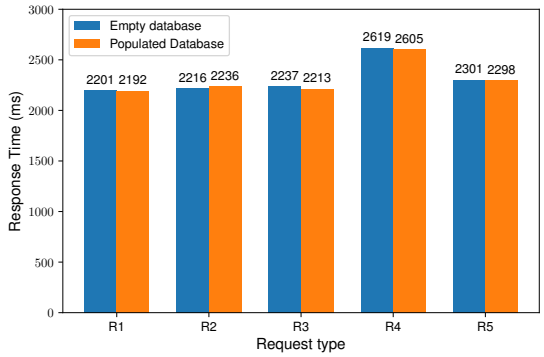
- Read Requests:
  - R1 - Get Asset by ID
  - R2 - Get Applied Tool by ID
- Write Requests:
  - R3 - Register Asset
  - R4 - Modify Asset
  - R5 - Register Applied Tool

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. 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. It is important to note that in experiments with larger core counts and larger RAM sizes, the throughput increased with increased concurrency values.

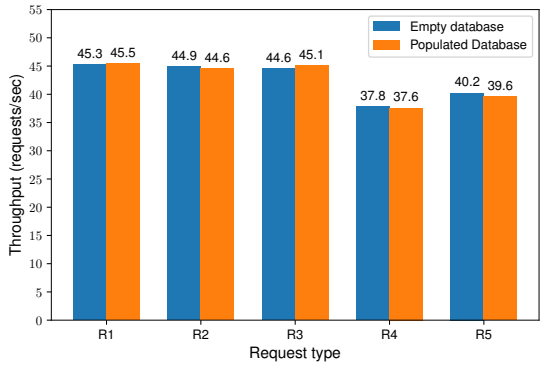
As we can observe from both graphics in Figure 2 and Figure 3, as expected there is a inverse correlation between response times and throughput: when the response time is greater, the throughput is lower. When repeating the tests with increased computing resources allocated to the Ledger, increasing the number of concurrent requests, the throughput values increased to above 200 requests per second. This

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



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

Fig. 2. Response times for different request types



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

Fig. 3. Throughput for different request types

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 Hyperledger Fabric such as [4], [6].

We can also observe in the graphics that both read and write methods achieve similar performances, both in response time and throughput. All the methods achieved similar metrics.

Methods with different complexity levels of logic programmed into the Smart Contract behaved similarly. This can be explained 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. 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.

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. 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 take considerably longer that these response times, 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 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 showed increased performance, that is in par with the general Ledger performance evaluations presented in other studies such as [4], [6]. 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.

VI. Conclusion

After the development of the solution, evaluation was carried, in a qualitative and quantitative perspectives. The solution was evaluated in a qualitative manner 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 infrastructure, 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.

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.

A. System Limitations and Future Work

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. As stated in Section V-F, 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 V-E, 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.

References

[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, ISBN: 9781450342186. DOI: 10.1145/2950290.2983952.

[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, ISBN: 9781450344272. DOI: 10.1145/2961111.2962627.

[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. DOI: 10.23919/CISTI52073.2021.9476488.

[4] N. Kannengießer, S. Lins, T. Dehling, and A. Sunyaev, "Trade-Offs between Distributed Ledger Technology Characteristics," *ACM Comput. Surv.*, vol. 53, no. 2, 2020, ISSN: 0360-0300. DOI: 10.1145/3379463.

[5] 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. DOI: 10.1109/ICSAI.2017.8248427.

[6] R. Nadir, "Comparative study of permissioned blockchain solutions for enterprises," in *2019 International Conference on Innovative Computing (ICIC)*, Nov. 2019, pp. 1–6. DOI: 10.1109/ICIC48496.2019.8966735.

[7] S. Nakamoto, "Bitcoin: A peer-to-peer electronic cash system," *Decentralized Business Review*, p. 21 260, 2008.

[8] 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, ISBN: 9781450355841. DOI: 10.1145/3190508.3190538.

[9] 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. DOI: 10.1109/BRAINS49436.2020.9223283.

[10] 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, ISSN: 2169-3536. DOI: 10.1109/ACCESS.2018.2870644.

[11] 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. DOI: 10.1109/ISS1.2019.8908031.

[12] 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. DOI: 10.1109/AQTR.2018.8402709.

[13] S. Rouhani, R. Belchior, R. Cruz, and R. Deters, *Distributed Attribute-Based Access Control System Using a Permissioned Blockchain*, 2020. arXiv: 2006.04384 [cs.CR].