University of
Stavanger

FACULTY OF SCIENCE AND TECHNOLOGY

MASTER'S THESIS

Study programme/specialisation: Computer Science	Spring semester, 2020 Open
Author: Jørgen Holme	
Programme coordinator: Rong Chunming Supervisor(s): Rong Chunming and Dhanya Therese Jose	
Title of master's thesis: Secure Distributed Computing Managed by Blockchain	
Credits: 30	
Keywords: Blockchain, Distributed Systems, Private Data, Hyperledger Fabric	Number of pages: 58 + supplemental material/other: Code on GitHub (Link provided) Stavanger, 04.07/2020

Faculty of Science and Technology
Department of Electrical Engineering and Computer Science

Secure Distributed Computing Managed by Blockchain

Master's Thesis in Computer Science
by

Jørgen Holme

Internal Supervisors

Rong Chunming

Dhanya Therese Jose

July 4, 2020

"Arguing that you don't care about the right to privacy because you have nothing to hide is no different than saying you don't care about free speech because you have nothing to say."

Edward Snowden

Abstract

Moving large amounts of data between networks for data analysis and computations presents several issues related to privacy and security. In collaboration with the TOTEM project [1], we propose a solution to these problems, by moving computations to the residence of the data. We introduce a novel approach for managing access to remote datasets and resources by blockchain technology through Hyperledger Fabric. Organizations with similar interests may join a consortium, which will form a private channel on the blockchain network, i.e., a separate ledger. Participating organizations will enroll their users, who thereafter must obtain a one-time-code using a smart contract in order to gain access to remote resources. We utilize Ansible for remotely deploying Hadoop clusters for computation, which will comprise several Docker containers. A user may run computations at several remote locations separately, and subsequently retrieve a combined result without having to share data between organizations. To ensure privacy between participating organizations we utilize chaincode and private data collections in Hyperledger Fabric. Finally, we demonstrate three ways of deploying the solution: locally, as a single cluster in the cloud using Azure, and across multiple clusters in the cloud using Azure. Our solution ensures data privacy by allowing data providing organizations to connect their own computational resources for data consumers to use. By running computations inside Docker containers on these resources, we ensure that these processes are isolated from the host system.

Acknowledgements

I would like to thank my supervisors Rong Chunming and Dhanya Therese Jose for their guidance throughout this thesis.

Contents

Abbreviations

VM	**V**irtual **M**achine
MSP	**M**embership **S**ervice **P**rovider
HDFS	**H**adoop **D**istributed **F**ile **S**ystem
IaC	**I**nfrastructure **a**s **C**ode
OTC	**O**ne **T**ime **C**ode
CA	**C**ertificate **A**uthority
PDC	**P**rivate **D**ata **C**ollection
AKS	**A**zure **K**ubernetes **S**ervice

Chapter 1

Introduction

1.1 Motivation

These days data is more valuable than ever. The tasks of storing and transferring data securely becomes increasingly difficult and expensive as the amount of data grows. Transferring such large datasets presents high costs and several security concerns. Furthermore, business leaders and data owners may be hesitant to transfer data which is considered to be private.

A solution for moving the computations to the data instead of vice versa satisfies the security and privacy concerns, as well as the extra expenses related to transferring large datasets between networks. This would allow the data to remain at its location or within its respective network at all times while still allowing remote users, such as an organization with shared interests, to access and perform computations on the data. Such computations may be performed to retrieve trivial statistical values, as well as more complex algorithms.

Furthermore, different organizations may possess datasets containing similar information. In such cases, it may be interesting to retrieve a combined computational result from these datasets. Traditionally, one could combine these datasets before performing computations, however, we aim to achieve the same result without sharing data between the participants.

We aim to find a solution satisfying these criteria. The TOTEM project [1] is seeking similar solutions as a part of their architecture, which we will discuss in more detail later.

1.2 Problem Definition

We seek a novel solution for securely and privately performing computations at the location of remote datasets. For this, we need to provision temporary computational environments at the site of the data. Inside these environments, computations need to be performed and results must be retrieved, while keeping the operations hidden from unauthorized entities.

Additionally, we face the issue of granting access to authorized users. Rules for controlling access to these resources must be agreed upon by the participating organizations, and allow access to authorized users without requiring interference or administration from the remote organizations. i.e., a user must be able to perform computations at any time, without having to wait for an administrator to grant them access.

Lastly, in the case of multiple datasets residing in different organizations, we must consider that the data is to be kept private between the participating organizations. Computational results from each participant must, therefore, be combined without exposure to other organizations.

Thus, the problem is essentially divided into three parts: How should we govern access to remote resources? How should we provision temporary computational environments at remote resources? How can we obtain a combined computational result from multiple datasets without sharing information between participating organizations?

1.3 Usecases/Examples

A project which would benefit from such a solution is the *Clarify* [2] project. Clarify is a multinational, multi-sectorial and multidisciplinary research and trainee program that comprises 12 early stage researchers from both engineering and medicine. The participants comprise mostly of universities and hospitals from Stavanger in Norway, Spain and the Netherlands. They aim to maximize the benefits of digital pathology and facilitate the daily work of pathologists by creating an automated digital diagnostic environment.

Clarify breaks down their goal into three parts. Firstly, advanced image processing techniques and AI methods for automatic WSI (whole-slide-image) interpretation for diagnosis and image retrieval. Next, novel cloud-oriented data infrastructure and algorithms for securely storing, retrieving and sharing a publicly available WSI database while assuring data interoperability and portability. Lastly, user friendly software such as computer-aided diagnosis and content-based image retrieval tools based on AI and cloud-computing algorithms.

We observe that their second goal resembles what we are trying to achieve in our thesis. Our solution may aid in their problem of securely storing, retrieving and sharing data by securely moving computations to the data, thereby making it unnecessary for data providers to share their data sets outside their respective networks. Furthermore, since the Clarify project pertains to medical data, it will surely be considered private. Our system will aid in keeping data private, while still allowing legitimate users to combine computational results harvested from the data.

1.4 Outline

Chapter 2

Here, we present some useful background knowledge, as well as the tools we use for developing our solution. Additionally, we compare different tools and select the most appropriate candidates.

Chapter 3

In chapter 3 we present some related works, provide a short summary of their work and explain how their approach differs from ours.

Chapter 4

Chapter 4 contains our solution approach. Here, we describe how we use the different tools and technologies presented in Chapter 2, and present our proposed architectures for solving our different problems.

Chapter 5

In this chapter we deploy our system in three different manners: Locally, on a single cluster in Azure using AKS, and across multiple clusters in Azure using AKS. We describe how we go about deploying the system, as well as what we need to consider when working with Azure and AKS.

Chapter 6

This chapter concludes our work. We summarize what we have achieved, as well as discuss possible future directions for the system.

Chapter 2

Background

2.1 Virtual Machines and Container Technology

2.1.1 Virtual Machines

A *Virtual Machine*[1], VM, is is used to emulate computer systems, and may be used to create isolated and fully functioning computer architectures within a computing environment, e.g., a computer or server. A *hypervisor* separates the environment's resources from its hardware and distributes them such that they can be used by the virtual machine. The physical hardware which holds the hypervisor is called the *host*, while the VMs which use its resources are called *guests*. Virtual machines are isolated from the rest of the system, which allows for several VMs to exist on one physical machine.

There are two types of hypervisors: Type 1 hypervisors schedule the VM resources directly to the hardware. An example of a type 1 hypervisor is a *Kernel-based Virtual Machine*, KVM. Type 2 hypervisors are hosted, meaning the VM resources are scheduled against a host operating system. An example of a type 2 hypervisor is *Oracle VirtualBox*.

Virtual machines allow for more efficient use of server resources by placing multiple virtual servers on one physical server to improve hardware utilization. This also provides redundancy and robustness since, in the case of a server failure, you have one or several other virtual servers running. VMs are also good for setting up testing and production environments, since they are isolated from the rest of the system.

[1]https://www.redhat.com/en/topics/virtualization/what-is-a-virtual-machine

2.1.2 Containers

Instead of virtualizing the entire computer system, containers virtualize the OS. *Container technology* ([3], [4]) can be used to run applications, with all their dependencies, isolated from other processes on the machine. Containers sit on top of the host OS to share the kernel as well as, in some cases, binaries and libraries. All shared resources are read-only, as to not interfere with other processes. Sharing these resources makes containers light-weight compared to VMs, since they do not need to reproduce the OS code, thus allowing a single OS installation to run several containers with ease.

There are several providers of container technology, such as Docker, rkt and Kubernetes, with one of the most popular choices being *Docker*. Docker offers light-weight, secure and portable containers, and is available for Linux, Mac OS and Windows Pro/Business. Docker containers can be created and destroyed in a matter of seconds, unlike VMs which could take several minutes. A major benefit of using Docker is the ease of running multi-container applications by using *docker compose*. Docker compose [5] is a tool which allows you to configure your application's services using a YAML file, and then create and start all the services with a single command.

The purpose of our thesis is to effectively provision temporary and isolated computing environments in which we will perform calculations. We consider this when choosing our method for isolating computations.

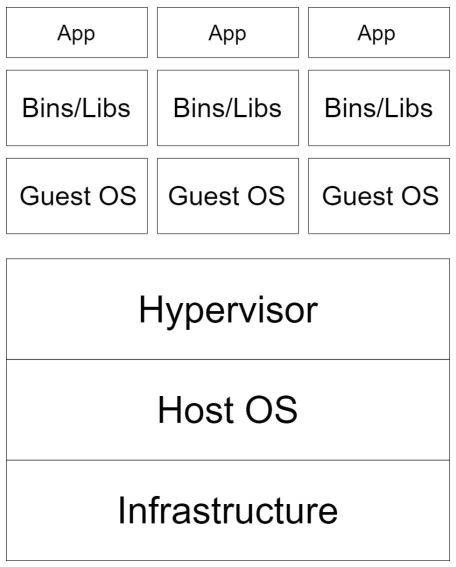
2.1.3 Virtual Machines vs. Containers

A figure illustrating the isolation differences between VMs and containers is shown in Figure 2.1[2].

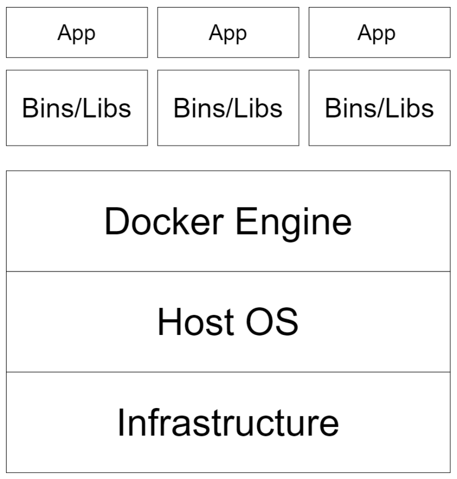
In a blog post by Mike Coleman [6], an employee at Docker, he uses the analogy of comparing VMs to houses, and containers to apartment buildings. In which, he proceeds to explain that houses are self-contained, meaning they have their own utilities such as electricity, plumbing and heating. They provide their own security, i.e., if your neighbour is careless and allows an intruder to break in, it should not matter to you. An apartment building, however, have a shared infrastructure for electricity, plumbing and heating. Also, a security slip by one of your apartment building neighbours could have consequences for you.

The same principals apply to VMs and containers. VMs have their own standalone OS and is virtualized at the hardware level, while containers shares the host OS, possibly along

[2] `https://www.backblaze.com/blog/vm-vs-containers/`



(a) Virtual Machines Diagram



(b) Docker Containers Diagram

Figure 2.1: Comparison between Virtual Machine and Docker Container structures

with other resources, and is isolated at the process level. Thus, the security threat within one virtual machine will not affect adjacent ones. However, a multi-container environment communicating over a shared network could potentially introduce several security threats. In such an environment, if someone gains unwanted access to a neighbouring container, it could have severe consequences for adjacent containers.

However, this does not mean that Docker container environments are insecure or unregulated, in fact, some argue that containers have security benefits over VMs[3]. One argument being that by dividing your application into microservices, each running in its own container with carefully defined interfaces, you are effectively decreasing the attack surface of your application.

Continuing Coleman's analogy of houses vs. apartment buildings, it is also worth noting that when buying a house you often risk buying more than you need, since houses usually come with a basic set of rooms, furniture, etc. Apartments, however, can be *bare-bone*, and only contain the minimum requirement of living. The same applies to VMs and containers. With Docker, you can create containers from images which contain only the essentials for a functioning environment, allowing you to only install what you need. On the contrary, virtual machines are often built with a full operating system, leaving it to the user to strip the system down to what they need.

In Table 2.1, an overview of some important differences are presented.

[3]https://thenewstack.io/thirteen-ways-containers-secure-virtual-machines/

Table 2.1: Virtual machines vs. Docker containers

Specs	Virtual machines	Docker containers
Host environment strain	Heavy-weight	Light-weight
Virtualization level	Hardware virtualization	OS virtualization
Startup time	Minutes	Seconds
Isolation level	Fully isolated	Process-level isolated

2.2 Tools and Concepts

2.2.1 Cloud Computing

Cloud computing [7] is a model for enabling omnipresent, convenient and on-demand access to a pool of computing resources. These resources can be rapidly provisioned with minimal management effort or service provider interaction.

Essential Characteristics

The cloud computing model comprises five essential characteristics: *On-demand self service*, *Broad network access*, *Resource pooling*, *Rapid elasticity* and *Measured service.*

- **On-demand self service**: A consumer can, when required, unilaterally and automatically provision computing capabilities without requiring human interaction with each service provider.
- **Broad network access**: All capabilities are available through standard mechanisms accessible through a variety of platforms (such as mobile phones, laptops, etc.).
- **Resource pooling**: Resources are served to several consumers using a multi-tenant model, where both physical and virtual resources are dynamically assigned depending on consumers' demand. Resources can refer to storage, processing, memory and network bandwidth.
- **Rapid elasticity**: Resources can rapidly scale both inward and outward depending on demand. Capabilities are dynamically provisioned, and are often seen as unlimited to consumers.
- **Measured service**: Cloud computing employs a metering capability, which is used to automatically control and optimize the resource usage. Transparency can be ensured for both the provider and consumer by monitoring, controlling and reporting resource usage.

Service Models

Cloud computing also employs three service models:

- **Software as a Service (SaaS)**: Providers may have applications running in their cloud infrastructure. These capabilities are available to the consumer through several interfaces such as a program interface or a web browser. The consumer does not control any of the underlying infrastructure or configuration related to the application, possibly, with the exception of some user-specific application settings.
- **Platform as a Service (PaaS)**: Consumers have the capability to deploy their own consumer-created or acquired applications as long as the programming language, libraries, services and tools are supported by the provider. The consumer does not control any of the underlying infrastructure, however, they control the application they have deployed and, possibly, some application-hosting environment settings.
- **Infrastructure as a Service (IaaS)**: Consumers have the capability to provision computing resources like processing, storage and networks where the consumer can deploy software which may include operating systems and applications. The consumer does not control any of the underlying infrastructure, however, they control the operating system, storage, deployed applications and, possibly, some networking components such as host firewalls.

Deployment Models

Furthermore, cloud computing comprises four deployment models.

- **Private cloud**: The cloud infrastructure is provisioned for a single organization. It may be owned, managed and operated by the organization, a third party or a combination of both. The infrastructure may exist on or off premises.
- **Community cloud**: The cloud infrastructure is provisioned for a community of organizations with shared interests. It may be owned, managed and operated by the entire, or parts of, the community, a third party or a combination of them.
- **Public cloud**: The cloud infrastructure is open to the general public. The infrastructure exists on the premises of the cloud provider.
- **Hybrid cloud**: The cloud infrastructure is based on a combination of two or more of the other deployment models.

2.2.2 Blockchain Technology

Blockchain [8] is a distributed digital ledger designed to be resilient against tampering. It allows for users to record any transaction made on the network in a shared and transparent manner, meaning that after a transaction has been made it cannot be changed, hence, resilience against tampering. This technology was first applied to cryptocurrency, namely *Bitcoin*, and was originally published in the paper *Bitcoin: A Peer to Peer Electronic Cash System* [9] under the pseudonym Satoshi Nakamoto.

There are four characteristics that make blockchain work as intended:

- **Ledger:** Blockchain uses an append-only ledger which provides the entire transactional history of the network. Transactions are appended as blocks, and unlike traditional databases, transactions recorded on the ledger cannot be overwritten. An illustration of how transaction blocks are connected is shown in Fig. 2.2.
- **Secure:** Blocks appended to the ledger contain the hash-value of its preceding block. This makes the blockchain cryptographically secure and ensures protection against data tampering.
- **Shared:** The ledger is shared by multiple participants to provide transparency across all participating nodes in the network.
- **Distributed:** The blockchain can be distributed across multiple nodes. By scaling up the number of participating nodes the network becomes more resilient to attacks by ill-intended participants. More nodes make it harder for such participants to impact the consensus protocol used by the blockchain network.

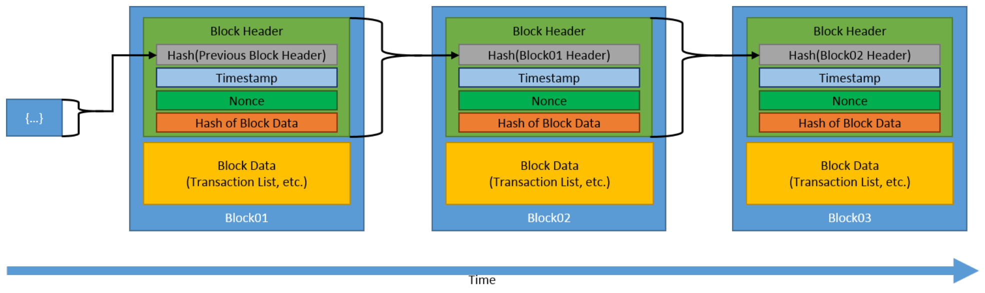


Figure 2.2: Illustration of a blockchain [10]

Blockchain networks can be divided into two categories: *Permissionless* and *permissioned*.

Permissionless Blockchain Networks

A blockchain network is *permissionless* if anyone can publish new blocks to the ledger without any authoritarian permission. Such platforms are often available as open-source software and may be downloaded by anyone. It follows that since anyone can publish blocks on the network, then anyone can transact on the blockchain as well as read and write to the ledger.

Furthermore, since a permissionless blockchain network is open to anyone, ill-intended participants may attempt to publish blocks that corrupt the system. This is prevented by introducing a *consensus* protocol, which requires users to spend or maintain some resources to be able to publish blocks. A consensus system usually rewards non-malicious behaviour by granting a native cryptocurrency to participants who conform to the protocol. The Bitcoin blockchain is an example of a permissionless blockchain network, rewarding those who conform to the *proof of work* consensus protocol with Bitcoin.

Permissioned Blockchain Networks

A *permissioned* blockchain network forces users to be to authorized by some authority, which may be centralized or decentralized. This means that such a network may allow anyone to read and write to the ledger and transact on the network, or it can restrict read and write access as well as who may submit transactions.

Permissioned blockchains can maintain the same traceability of digital assets as permissionless blockchains, as well as the same distributed and resilient data storage. Furthermore, permissioned blockchains also use consensus models for publishing blocks. However, because participants have to be authorized before joining the network, and therefore have a level of trust between them, there is often no need to base the consensus model on spending or maintaining any resources.

An example of where permissioned blockchains are especially useful is when organizations want to work together but do not fully trust each other. They may want to control or protect their data and resources for some reason, while still being able to cooperate with their business partners. The participating organizations, often referred to as a *consortium*, may then establish a permissioned blockchain network, agree on which consensus model they want to use, and authorize the appropriate users. This provides trust as well as transparency which may help with business decisions as well as holding malicious participants accountable.

Lastly, permissioned networks can provide a level of transaction privacy by only allowing certain users to view transaction information based on their identity or credentials.

2.2.3 Smart Contracts

Smart contracts are a supplement to blockchains which allows user-defined rules and restrictions for transactions. The involved parties will agree on the rules of a transaction just as with traditional contracts, however, the rules are defined in code. These smart contracts will then execute every time a transaction is made to make sure the defined rules are followed.

This concept was first introduced by Nick Szabo in 1994 [11].

2.2.4 Ethereum

We previously mentioned Bitcoin as an example of permissionless blockchains. However, the Bitcoin blockchain is only fit for transacting the cryptocurrency known as Bitcoin. Another permissionless blockchain technology called *Ethereum* [12] utilizes blockchains for more than just monetary value by exploiting smart contracts. They set out to create a blockchain with a built-in programming language to be used for creating smart contracts, allowing users to develop systems covering a vast selection of use cases.

Despite the advantages of using smart contracts, it introduces some security concerns. Malicious users may deploy smart contracts with infinite loops or very computationally heavy code. This can be exploited to perform Denial-of-Service attacks (DoS attacks), halt the network, and deny access to legitimate users. Ethereum tackles this issue by introducing *gas*. A user has a certain amount of gas available, and when they wish to run any smart contract on the blockchain they must have a sufficient amount of gas. If a user's gas runs out during a computation, it will stop running the code. This mechanism allows the Ethereum blockchain to be permissionless and utilize smart contracts, without being victim to DoS attacks.

2.2.5 Hyperledger Fabric

Hyperledger is an open-source project created to enhance blockchain for enterprises. The project started in 2015, hosted by the Linux Foundation, and is a collaborative effort between many different companies. It comprises over 230 organizations and several projects, including IBM's *Hyperledger Fabric*.

Hyperledger Fabric ([13], [14]) is an enterprise-grade permissioned distributed ledger framework created by IBM. It focuses on a modular and configurable architecture, allowing it to meet the requirements of many different use cases such as banking, insurance, healthcare, etc. Fabric supports smart contracts written in general purpose programming languages including Java, Go and Node.js, meaning users do not need to learn a domain-specific language to write them. Smart contracts are called *chaincode* in Hyperledger Fabric, this is further discussed in Section 2.2.5.

Furthermore, Fabric is a permissioned ledger (recall Section 2.2.2) meaning organizations who wish to transact privately without completely trusting each other can do so on a Fabric network. Organizations may agree how the access to data and resources should be governed before any transactions take place, due to the modularity and configurability of Fabric. A feature which demonstrates this configurability, is the *pluggable consensus protocol*. This allows for participating organizations to customize the platform to fit their specific use case. For example, a network comprising a single enterprise versus a network with several competing organizations will have different needs in terms of how comprehensive the consensus protocol is.

Channels

As discussed in Section 2.2.2, permissioned blockchains force users to be authorized before joining the network. Hyperledger Fabric allows for privacy and confidentiality through *channels*. A channel is formed by a consortium of organizations, which share a separate channel ledger and are free to transact as long as they conform to the policies defined on the channel. This allows for transparency between the members of the consortium, while still keeping their transactions private from outsiders. Note that the channel we describe here is known as an *application channel*. This differs from the *system channel*, which controls the configuration of the Fabric network. In this thesis, we refer to application channels when mentioning channels.

A channel ledger will comprise a *world state* and a *transaction log*. The world state represents the current state of the channel ledger, while the transaction log is the history of transactions which has lead to the current world state, i.e., the transaction log is the blockchain. A channel will also logically host smart contracts, which in Fabric are written in *chaincode*, and may be invoked by applications who wish to interact with the ledger.

Peers and Orderers

The nodes which comprise a Hyperledger Fabric network are primarily *peer nodes* and *orderer nodes*, who cooperate to ensure that only proper transactions are committed to the ledger. Peers may take on different roles in the network, however, for now it is enough to know that some peers act as *endorsing peers*, which will endorse a transaction before sending it to the orderer. The following steps are taken to commit a transaction to the ledger.

1. A transaction proposal is sent to each endorsing peer, who will run and subsequently endorse the transaction before sending it to an orderer.
2. The orderer will ensure that the transaction is endorsed by the necessary peers. It will then add the transaction to the next block and distribute it to all peers participating on the channel.
3. Each peer will then inspect the block to validate that every peer has received the same result. Upon a successful validation, the peers will commit the block to the ledger.

Every peer will additionally host a ledger instance for each channel it is participating in. Furthermore, if the peer is an endorsing peer, it will host an installation of every chaincode instantiated on the channels. We discuss chaincode further in Section 2.2.5.

Membership Service Provider

Members of the network and channels are enrolled through a *Membership Service Provider* (MSP). The MSP maps a user's certificate to the organization it is a member of. Thus, the MSP can turn the *identity* of a user into a *role*. Organizations can agree on the permissions a role should be granted on the network or in a channel. Organizations may, therefore, govern their data and resources by establishing MSPs which determine the access level roles should be allowed, as well as the operations they can perform.

Chaincode

The terms *smart contract* and *chaincode*[4] are often used interchangeably. One could define smart contracts as the transactional logic for interacting with the world state, which is then packaged into a chaincode and deployed to the Fabric network. For simplicity's sake, we will think of a chaincode as Fabric terminology for a smart contract. At the time of writing, Fabric supports chaincode written in *Go*, *Java* and *Node.js*.

Chaincodes are deployed on channels to put, get and delete states in the world state. We may also leverage chaincode to generate and return values to the user. Different applications of chaincode are discussed in Section 4.3.3. When deploying a chaincode

[4]`https://hyperledger-fabric.readthedocs.io/en/release-1.4/smartcontract/smartcontract.html`

to a channel, it must first be *installed* on the endorsing peers from each organization participating on the channel. Additionally, the chaincode must be *instantiated* on the channel by one of the participating peers. Therefore, we may say that a chaincode is *physically hosted* on peers, while it is *logically hosted* on the channel.

Hyperledger Fabric Example Network

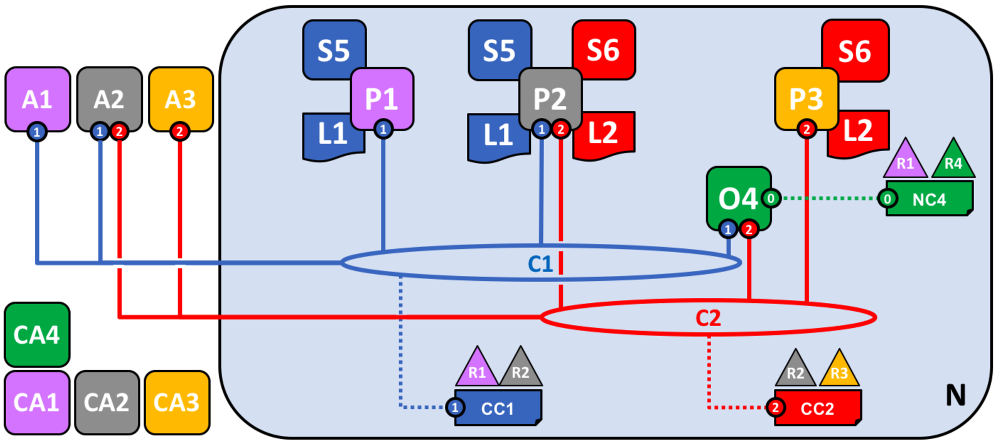


Figure 2.3: Illustration of a Hyperledger Fabric network [15]

An illustration of a Hyperledger Fabric network comprising four organizations (R1, R2, R3 and R4) is shown in Fig. 2.3. Here, we observe *peers* (P1, P2 and P3) belonging to three different organizations. These peers physically host smart contracts (S5 and/or S6), as well as a channel ledger copy (L1 and/or L2) of the channel(s) they are a member of. We observe two channels (C1 and C2) who each have their own separate channel configuration (CC1 and CC2) defined by the member organizations. The external applications (A1, A2 and A3) can interact with the channel ledgers by invoking the smart contracts which are hosted on them. Each organization has their own *certificate authority* (CA1, CA2, CA3 and CA4) which issue X.509 certificates for authenticating users. Lastly, there exists an *orderer* (O4) which is defined by the network configuration (NC4).

2.2.6 Hadoop and Big Data

Big data [16] is a term used to describe data sets which are too large to be handled using conventional mechanisms. Important characteristics of big data are *volume*, *velocity*, *variety* and *variability*. New techniques for data handling are required to work with big data, such as parallel data processing. One framework which allows for parallel and distributed processing of large data sets is *Hadoop*.

Apache Hadoop [17] is a framework for parallel processing of large data sets in a distributed setting. Processing tasks are distributed across clusters of computers who each contribute their own storage and computational power. Hadoop is designed to be highly scalable, and is programmed to detect and handle failures at the application layer, thereby ensuring availability. The framework comprises several modules, the two main layers are *Hadoop MapReduce* and *Hadoop Distributed File System* (HDFS).

HDFS is a distributed file system which consists of a master-slave architecture. A HDFS cluster comprises a single *NameNode* which acts as the master server that manages the file system namespace and regulates file access, as well as several *DataNodes*. After a file has been divided into fixed-size blocks, they are distributed among these DataNodes. The NameNode is tasked with mapping these blocks to the DataNodes, as well as performing namespace operations such as opening, closing and renaming files and directories. Furthermore, the NameNode can instruct the DataNodes to create, delete and replicate blocks. It is the DataNodes' job to handle read and write operations from the client of the file system.

Hadoop provides solutions for moving computations instead of moving data, which directly addresses the motivation for our thesis. Therefore, we choose Hadoop as our computational framework.

2.3 Choosing an Automated Deployment Tool

There exist several tools for automating IT infrastructure and application-deployment. We will investigate some advantages and drawbacks to see which is most fitting for our problem.

2.3.1 Infrastructure as Code

Infrastructure as Code[5], IaC, is the concept of managing infrastructure as a descriptive model. The code can be imperative, meaning an ordered list of instructions which walks through the configuration step by step, or declarative, meaning we define the desired final state of our environment.

IaC tackles the issue of *environment drift*, which is the problem of each configuration in an environment being unique, and hard to exactly replicate due to manually maintaining all node configurations. Inconsistencies in an environment can lead to issues during

[5] https://docs.microsoft.com/en-us/azure/devops/learn/what-is-infrastructure-as-code

deployment. IaC solves this by assuring that a set of nodes can consistently run the same configuration by deploying the same model on each of them.

We will be using the concept of IaC to deploy Docker containers on a set of resources, and within them, run the necessary computations. Next, we must choose the tool for which we will use to provision these Docker containers. There exist several viable choices, of which we consider two: *Ansible* and *Puppet*.

2.3.2 Ansible

Ansible[6] is an open-source software which automates the process of IT infrastructure and application-deployment. An Ansible-managed infrastructure will consist of one or several *control nodes*, which will have Ansible installed on them, and *managed nodes* which will receive instructions from the control nodes. Ansible holds the advantage of not requiring any client-side installation, as it is based on a *push configuration* where a control node pushes out tasks to its managed nodes.

The tasks which are to be executed on the managed nodes can be defined singularly ad-hoc, or as a series of tasks in a *playbook*. Playbooks are written in *YAML*, making them both easy to define and easy to read.

To ensure a secure connection between a control node and a managed node, Ansible uses SSH with public key authentication. This means the managed node must grant access to the control node before any commands can be pushed.

2.3.3 Puppet

Puppet[7] is another tool which utilizes IaC. This infrastructure consists of several *Puppet agents* which are controlled by a *Puppet master*. Puppet is based on a *pull configuration*, unlike Ansible, which requires an installation of the software on both master and agent nodes.

Puppet uses its own declarative language to create files called *manifests*, which are used to describe the desired state of a system. Furthermore, Puppet compiles these manifests into *catalogs*, which describes the desired state of a specific node. This language holds the advantage of being generally easier to debug than YAML, however, YAML is easier to understand and results in human-readable playbooks.

[6] https://www.ansible.com/overview/it-automation
[7] https://puppet.com/docs/puppet/latest/puppet_overview.html#puppet_overview

The ease of setup and installation, as well as the practicality of a push configuration makes Ansible the best choice for our temporary container deployments.

Chapter 3

Related Work

3.1 The TOTEM Project

TOTEM [1], token for controlled computation, aims to integrate blockchain technology with big data to move computations to the data. Their proposed architecture utilizes a Hyperledger Fabric network for governing access to remote resources, and the Hadoop framework for running computations at the residence of data sets.

Furthermore, the project proposes an entity called TOTEM, which is used to prevent ill-intending users from executing malicious code by putting constraints on the computational code submitted by users. A user will obtain a pre-defined totem value which will be gradually exhausted as the operation code (opcode) comprising the computational code is executed. When a user's totem value is spent, they will no longer be able to run their code on the remote data sets. In the TOTEM project's current architecture, the totem value resembles Ethereum's gas concept [12], as discussed in Section 2.2.4.

In order to determine when a user has exhausted their totem value, they employ a system for estimating the computational cost of opcode called a *totem estimator table*. The computational code is submitted to the Hyperledger Fabric blockchain, where a smart contract resides to determine the cost of each opcode.

In the TOTEM project's *customised computational framework* they introduce a master node which communicates with a *totem manager*, and several slave nodes which communicate with *totem updaters*. The totem manager receives the user's available totem, as well as the estimated totem to perform the desired computation. It is then the totem manager's job to calculate the usage of the totem in between each executed opcode, while updating the master node on whether or not computations should continue.

Our proposed solution is in association with the TOTEM project. However, the usage of totem values and the customised computational framework is beyond the scope of this thesis. We aim to implement solutions for resource governance, temporary computational environments at remote resources, and secure result retrieval in a multi-provider scenario. These solutions may then be implemented into TOTEM's proposed architecture.

3.2 Data Privacy and Blockchain

Systems built around a centralized architecture are often forced to trust a third party with their information, which presents issues when data owners are reluctant to share their information. Aitzhan and Svetinovic [18] aims to solve these security and privacy issues in energy trading, by proposing a decentralized system using multi-signatures, blockchain and anonymous messaging streams. Their system is built upon the Bitcoin blockchain [19] to eliminate the need of trusted third parties, and utilize Bitmessage [20] for propagating encrypted data in messaging streams. However, their solution is based on a trust-less scenario of strangers trading with each other and, being built upon the Bitcoin blockchain, entails a permissionless system. Our system proposes the use of a Hyperledger Fabric, a permissioned blockchain, to keep information private between organizations with shared interests, as well as *private data collections* (discussed further in 4.5.1), for keeping private data hidden between organizations.

Brandenburger et al. [21] discusses the problems of keeping data private on blockchains. Smart contracts cannot keep secrets as their data is replicated on all nodes. Furthermore, they investigate the pitfalls of combining blockchain with *trusted execution environments* (TEEs). A TEE, such as Intel's Software Guard Extensions[1] (Intel SGE), will isolate its executions on the host CPU, meaning the host's environment does not need to be trusted and data is kept private. They go on to mention the susceptibility of rollback attacks in this approach. Finally, they propose a solution using Intel SGE with Hyperledger Fabric for securely executing chaincode in an isolated environment.

In our case, we may trust the environment in which we run our computations, since the data provider will connect their own computational environment. Therefore, there is no need for a TEE. Furthermore, we need to allow combining computational results from multiple data providers while keeping data and results private from other organizations in the consortium. As mentioned, we use private data collections to solve this.

Moreover, Brandenburger et al. [21] raises an interesting point when discussing the possibility of using cryptographic protocols such as multiparty secure computations as

[1]https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html

a solution approach. They point out that these approaches are not mature enough to easily handle general-purpose computations. Our approach of running computations individually at each data provider and retrieving results with private data collections allows us more freedom in what computations we can run.

Benhamouda et al. [22] address the problem of private data on Hyperledger Fabric. They point out the fact that all peers on a channel must have the same view of the world state, meaning all data and transactions are transparent to all participants. Since this may be undesirable in several scenarios, they propose encrypting private data before committing it to the blockchain, and utilize a multi-party computation solution when private data is needed in a transaction. They introduce two new components which should be added to the Hyperledger Fabric architecture to support their solution. The first component they introduce would allow for a *local configuration*, meaning the chaincode running at peers with private data should have access to parameters which are not available to other peers. The second component addresses *inter-peer communication*, which entails that private data may interfere with the endorsement decision for peers who do not see that data.

We solve these issues by using private data collections in Hyperledger Fabric. Here, we can introduce and utilize collections of private data directly in our chaincode, by giving these collections their own policy regarding who may access them. Private data is then communicated between authorized peers using a gossip protocol and stored in a separate private database, with no involvement from the orderer. The concept of private data collections and how they function are discussed in Section 4.5.1. By exploiting this concept, we do not need to introduce any additional components to Hyperledger Fabric, nor do we need any multi-party computation implementations.

Chapter 4

Solution Approach

4.1 Introduction

In Section 1.2 we present three problems that need to be considered. To ensure that connections and permissions are granted safely, we will connect to a Hyperledger Fabric network, which will return a *one-time-code* (OTC) to users with proper permissions. Furthermore, we choose to use Ansible to provision Docker containers, in which computations will be run inside a Hadoop cluster. Valid users may use their obtained OTC to gain access to the remote resources, and subsequently provision their temporary infrastructure with Ansible. Lastly, we utilize private data collections in Hyperledger Fabric for keeping data and computational results private between participating organizations.

4.2 Analysis

As per TOTEM's [1] proposed architecture, we identify the entities participating in our network as *data providers* and *data consumers*. These can be defined as follows:

Definition 1 (**Data Consumer**): *A member of the Hyperledger Fabric network who wishes to run their computational code on a data provider's dataset, using the provider's resources.*

Definition 2 (**Data Provider**): *A member of the Hyperledger Fabric network who provides a dataset and a set of resources which may be used by a data consumer to run computations.*

Observe that data consumers and data providers correspond to control nodes and managed nodes from the perspective of Ansible.

Our goal is to automate the process of performing computations on data providers' data sets, possibly containing private information, by bringing the computations to the data. However, before any computation take place, we must create a system for granting access to data providers' resources and data. First, we must first define what granting access entails in our scenario, and the steps a user must complete to obtain access.

As mentioned, our proposed solution uses Ansible for provisioning the necessary infrastructure and running Hadoop jobs. Since Ansible relies on SSH connectivity to push commands, we need to implement a system which grants SSH access to data consumers. Also, we assume that a user is authorized before attempting to access resources. An authorized user refers to someone who is enrolled in the Hyperledger Fabric network as a member of a participating organization.

When access has been granted to properly authorized users, they are free to run the Ansible playbooks which will automatically install, start and utilize Docker to run MapReduce jobs in Hadoop. After the computations have completed, we are faced with the challenge of handling the results. We differ between two scenarios when retrieving results. These can be defined as follows:

Definition 3 (**Single-Provider Scenario**): *When a data consumer runs computations on the data set of one single provider.*

Definition 4 (**Multi-Provider Scenario**): *When a data consumer runs computations on several data sets from several data providers.*

The latter of these two scenarios presents challenges regarding privacy. We must implement a system for combining and retrieving results without sharing any data with other organizations on the channel.

Our final system will comprise three peer organizations and one orderer organization. We take inspiration from the Clarify project, mentioned in Section 1.3, and create the consortium in Table 4.1.

Table 4.1: Consortium of Participating Organizations

Name	Type	Count
Stavanger	Peer Organization	1
Netherlands	Peer Organization	1
Spain	Peer Organization	1

We will use this consortium to form a channel on our network, which we will use to demonstrate our solutions. Additionally, we add an orderer organization simply called *OrdererOrg*, which will comprise a single orderer on our network.

The rest of this chapter will comprise sections describing how we solve the different challenges of our thesis. The sections are structured as follows:

- Governing resource and data access using Hyperledger Fabric.
- Running computations with Ansible playbooks, Docker and Hadoop.
- Private data collections for retrieving distributed private results in a multi-provider scenario

4.2.1 Assumptions

We implement our solution as a proof-of-concept, meaning we acknowledge that the system contains several caveats and is not a production-ready solution. Therefore, we must make some assumptions about the environment we are working in.

Firstly, we assume that data providers are running Linux in their computational resources, namely, we are developing the system for Ubuntu systems. Furthermore, we assume that the computational code is already present at the provider's residence. In TOTEM's proposed architecture, the computational code will be written in a custom SDK and transferred using the blockchain, however, this is beyond the scope of this thesis.

We also acknowledge the vulnerability of a man-in-the-middle attack[1] when transporting the OTC and public key from the data consumer to the data provider. We will assume a secure communication when exchanging the public key and OTC, however, this issue should be addressed in future works, and is discussed further in Section 6.2.

[1] `https://en.wikipedia.org/wiki/Man-in-the-middle_attack`

4.3 Governing Resource and Data Access using Hyperledger Fabric

We present a proposed architecture for data and resource governance managed by Hyperledger Fabric.

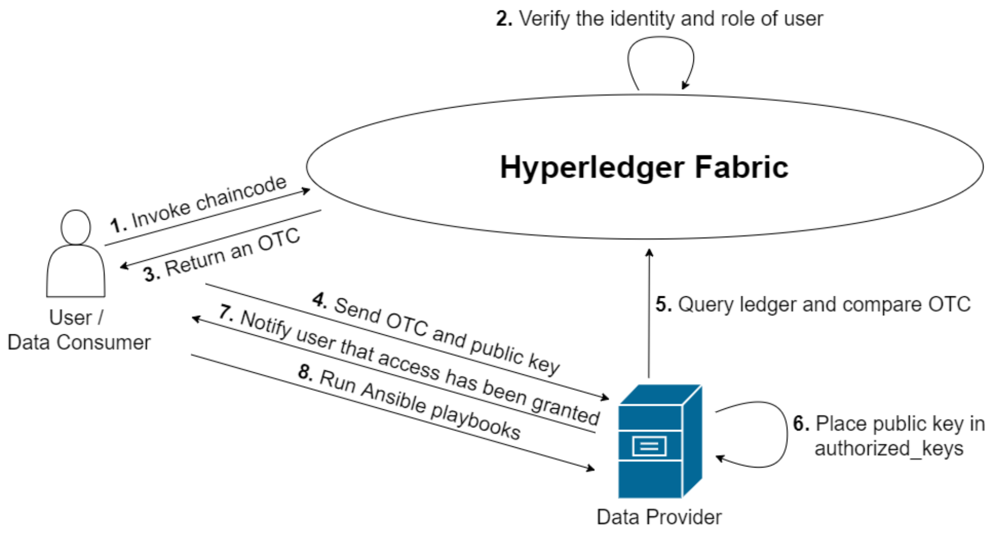

2. Verify the identity and role of user
Hyperledger Fabric
1. Invoke chaincode
3. Return an OTC
User / Data Consumer
4. Send OTC and public key
7. Notify user that access has been granted
8. Run Ansible playbooks
5. Query ledger and compare OTC
6. Place public key in authorized_keys
Data Provider

Figure 4.1: Proposed Architecture for Authentication Managed by Hyperledger Fabric

In Fig. 4.1 we observe the process one must go through in order to gain access to a data provider's resources. A closer look into the operations happening inside the Hyperledger Fabric network and the data provider's computational resource are discussed later in this section.

1) The data consumer invokes the `grantAccess` chaincode which will 2) check the identity of the consumer, and make sure they have the appropriate *role* to perform the operation (discussed further in Section 4.3.1). 3) Upon confirming the privilege of the data consumer, the chaincode will return a one-time-code[2] (OTC). In short, an OTC is a value which may be used to authenticate a user for a single session. 4) The data consumer will subsequently send their OTC and public key to the provider, which will 5) query the Hyperledger Fabric network to receive the OTC that is recorded on the ledger. If the OTC received by the alleged consumer matches the OTC recorded on the ledger, the provider will 6) place the consumer's public key in its `authorized_keys`, and 7) notify the consumer that access has been granted. The data consumer now has access to the data provider's resources, and may 8) push Ansible commands for provisioning infrastructure and running computations.

[2]https://en.wikipedia.org/wiki/One-time_authorization_code

Before implementing this authentication system we must launch a Fabric network, as well as write the necessary chaincode and client scripts for realizing the proposed functionality. Firstly, we discuss the process of setting up a Hyperledger Fabric network.

4.3.1 Configuring and Launching a Hyperledger Fabric Network

We want a system which allows all participating parties to agree on how resources and data should be governed. This makes *Hyperledger Fabric* a fitting choice as our tool for governance.

For our purpose, we will first launch a network containing three peer organizations and one orderer organization. The network will comprise a single orderer and a single peer in each organization. In a production environment it would be reasonable to deploy several orderers and peers per organization, however, for testing our proposed solution it is sufficient to deploy a minimal network configuration.

Each organization will decide a *policy*[3] for which of their users may read, write and administer in the network. The definition of these policies support combinations of `AND`, `OR` and `NOutOf`. For example, and organization policy stating that a writer can be either an admin or a client would be expressed as Listing 4.1.

```
Writers:
    Type: Signature
    Rule: "OR('Org1MSP.admin', 'Org1MSP.client')"
```

Listing 4.1: Organization Policy Example

The `Type:Signature` field refers to the fact that a writing operation requires the signature to be compliant with the `Rule` that is defined. In this case the policy definition is based on the *role* of the user, which is issued by the MSP. As previously mentioned, a user is given an X.509 certificate[4] by a Certificate Authority (CA), which the MSP will use to assign a role to the user. We use the CA issued by Hyperledger Fabric, called Fabric-CA, which allows users to be registered with admin, peer, client, orderer, or member as their role.

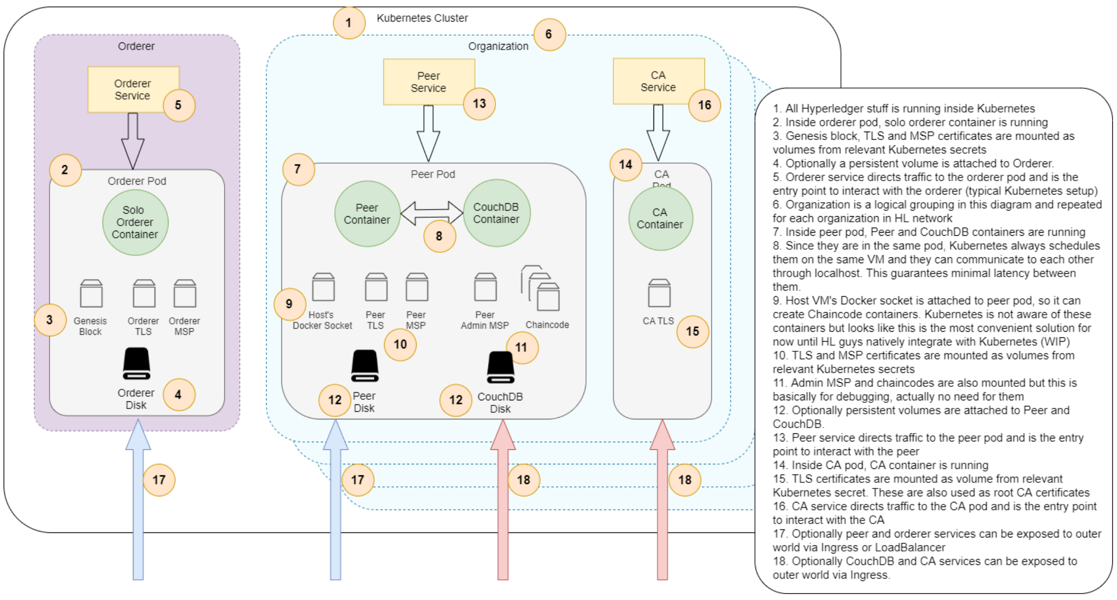
Furthermore, we must define the channels which will populate our network, and which peer organizations will join them. The channel will also enforce a policy to decide who can perform which operations. We describe how the channel is configured in Section 4.3.2.

[3] `https://hyperledger-fabric.readthedocs.io/en/release-1.4/policies/policies.html`
[4] `https://www.ssl.com/faqs/what-is-an-x-509-certificate/`

4.3.2 PIVT

The task of configuring and setting up a Hyperledger Fabric network from scratch is a time-consuming and error-prone task. Luckily, there are tools which facilitate network configurations as well as network interactions.

We utilize *PIVT* [23], which is a tool for running and operating Hyperledger Fabric v1.4.5 in *Kubernetes*. Kubernetes[5] is an open-source system for orchestrating containers, which automates deployment, scaling and management of container applications. Kubernetes leverages *services* which directs traffic to *pods*, which again consists of several containers. For example, in the case of Hyperledger Fabric we will have a peer service, which acts as an entry point for interactions with peers. This service directs traffic to a peer pod, which will consist of one or several peer containers. We discuss Kuberentes, and launching a Hyperledger Fabric network in Chapter 5. An extensive overview of a simple Hyperledger Fabric network in Kubernetes, from the PIVT GitHub repository [23], is shown in Fig 4.2.



1. All Hyperledger stuff is running inside Kubernetes
2. Inside orderer pod, solo orderer container is running
3. Genesis block, TLS and MSP certificates are mounted as volumes from relevant Kubernetes secrets
4. Optionally a persistent volume is attached to Orderer.
5. Orderer service directs traffic to the orderer pod and is the entry point to interact with the orderer (typical Kubernetes setup)
6. Organization is a logical grouping in this diagram and repeated for each organization in HL network
7. Inside peer pod, Peer and CouchDB containers are running
8. Since they are in the same pod, Kubernetes always schedules them on the same VM and they can communicate to each other through localhost. This guarantees minimal latency between them.
9. Host VM's Docker socket is attached to peer pod, so it can create Chaincode containers. Kubernetes is not aware of these containers but looks like this is the most convenient solution for now until HL guys natively integrate with Kubernetes (WIP)
10. TLS and MSP certificates are mounted as volumes from relevant Kubernetes secrets
11. Admin MSP and chaincodes are also mounted but this is basically for debugging, actually no need for them
12. Optionally persistent volumes are attached to Peer and CouchDB.
13. Peer service directs traffic to the peer pod and is the entry point to interact with the peer
14. Inside CA pod, CA container is running
15. TLS certificates are mounted as volume from relevant Kubernetes secret. These are also used as root CA certificates
16. CA service directs traffic to the CA pod and is the entry point to interact with the CA
17. Optionally peer and orderer services can be exposed to outer world via Ingress or LoadBalancer
18. Optionally CouchDB and CA services can be exposed to outer world via Ingress.

Figure 4.2: A Simple Hyperledger Fabric Architecture in Kubernetes

As per the Hyperledger Fabric docs and its samples, we run our network in a Docker container environment, which we will orchestrate using Kubernetes. PIVT provides *Helm charts* to facilitate launching a Fabric network, as well as interacting with it. Helm[6] is a package manager for Kubernetes, which manages charts. Charts are packages of pre-configured Kubernetes resources, which helps us define, install and update Kubernetes applications.

[5] https://kubernetes.io/docs/home/
[6] https://helm.sh/

PIVT is a collaborative effort between APG[7] and Accenture NL[8]. According to their GitHub repository they provide Helm charts for:

- Configuring and launching a Hyperledger Fabric network.
- Populating the network declaratively with channels, peers and chaincode.
- Adding new peers to running networks and updating channel configurations declaratively.
- Backing up and restoring the state of the network.

Being able to run these operations declaratively means they are safe to run multiple times, e.g, a channel will not be created if it is already present.

Before we can use these Helm charts, we need to satisfy some prerequisites. Firstly, PIVT requires that we have a running Kubernetes cluster, on which we will launch our network. For testing purposes we utilize *Minikube*[9], which launches a single-node Kubernetes cluster inside a VM on our local machine. Next, we need the binaries for Hyperledger Fabric, Helm, *jq*[10] and *yq*[11]. In short, yq will take yaml as input, convert it to json, and pipe it to jq, which is a lightweight json-processor. Lastly, we will need *Argo Workflows*[12], a container-native workflow engine for orchestrating parallel jobs in Kubernetes.

After we have installed all requirements, we may use PIVT to launch our network. For this, there are three files we must configure:

`configtx.yaml`, `crypto-config.yaml` and `network.yaml`.

The `network.yaml` defines the channels we will add to the network, which peer organizations will join the channels, as well as what chaincode will be installed on the channels. Our `network.yaml` definition is shown in Listing 4.2. In this definition, we first observe a `genesisProfile` and `systemChannelID`. These are both used by PIVT to create the *genesis block*, which is the first block added to our blockchain, containing configuration information. As mentioned in Section 2.2.5, the system channel controls the configuration of our network, and the `systemChannelID` is used to identify this channel.

Next, we define the `channels`, containing the channel names which will populate the network, as well as the peer organizations which will participate in them. The final section,

[7]https://www.apg.nl/en
[8]https://www.accenture.com/nl-en
[9]https://kubernetes.io/docs/setup/learning-environment/minikube/
[10]https://stedolan.github.io/jq/
[11]https://pypi.org/project/yq/
[12]https://argoproj.github.io/projects/argo/

```
network:
  genesisProfile:  OrdererGenesis
  systemChannelID: ansiblenetwork

  channels:
    - name: common
      orgs: [Spain, Netherlands, Stavanger]

  chaincodes:
    # CHAINCODE FOR ACCESSING REMOTE RESOURCES.
    - name: access-chaincode
      version: # "2.0"
      orgs: [Spain, Netherlands, Stavanger]
      channels:
      - name: common
        orgs: [Spain, Netherlands, Stavanger]
        policy: OR('SpainMSP.member', 'NetherlandsMSP.member',
                'StavangerMSP.member')
```

Listing 4.2: Network Definition

`chaincodes`, describes which chaincodes should be included, and to which channels they should be installed. In our case, we define a channel called *common*, which will be joined by all of our peer organizations, i.e., Spain, Netherlands and Stavanger. We also specify that the `access-chaincode` shall be installed on peers in all organizations, with the policy that members in any of the organizations may invoke the chaincode. In a real-world scenario, the chaincode will be deployed as a collaborative effort between all data providers. This means that all providers must agree on the chaincode that will govern their resources, and the policies deciding who gets to invoke the chaincode.

Next, we inspect the `crypto-config.yaml` file. The file is shown in Listing 4.3. Here, we first observe the `OrdererOrgs` section, which lists the orderer organizations in the network, for which we specify a name, domain and hostname. Additionally, we set the `EnableNodeOUs` parameter to true. This tells the MSPs in the network to enable the use of roles.

The final section, `PeerOrgs`, specify the peer organizations participating in the network. Here, we also specify name, domain and the use of roles. Also, we use the `Template` and `Count` parameters to specify the number of peers each organization should contain. This file is used by PIVT to generate the cryptographic material for each organization, such as MSP directories, certificates and keys.

It is important to note that while channels and organizations are specified in `crypto-config.yaml` and `network.yaml`, this is not where they are defined. The bulk of the network is defined in the `configtx.yaml` file. Here, we define each orderer and

```
OrdererOrgs:
  - Name: OrdererOrg
    Domain: ordererOrg.com
    EnableNodeOUs: true
    Specs:
      - Hostname: orderer0

PeerOrgs:
  - Name: Stavanger
    Domain: stavanger.no
    EnableNodeOUs: true
    Template:
      Count: 1
    Users:
      Count: 1

  - Name: Netherlands
    Domain: netherlands.nl
    EnableNodeOUs: true
    Template:
      Count: 1
    Users:
      Count: 1

  - Name: Spain
    Domain: spain.es
    EnableNodeOUs: true
    Template:
      Count: 1
    Users:
      Count: 1
```

Listing 4.3: Crypto-Config Definition

peer organization, along with all organization policies, channels and applications. This is also where we define consortiums, which subsequently are used to form channels.

An extract from our `configtx.yaml` file, showing the definition of the Spain organization, is shown in Listing 4.4. Refer to Appendix A for the whole file. The name of the organization is defined in the very first line, followed by the name, ID and file path of the organization's MSP. Next, we observe the organization policies which determine reading, writing and administrative rights of organization members. Lastly, we define an *anchor peer*. As mentioned in 2.2.5, peers may take on different roles, one of which being the anchor peer, which is used for cross-organization gossip communication. We discuss gossip communication in Section 4.5.1.

The three files mentioned are the basic building blocks for setting up a Hyperledger Fabric

```
- &Spain
    Name: SpainMSP

    ID: SpainMSP

    MSPDir: crypto-config/peerOrganizations/spain.es/msp

    Policies:
        Readers:
          Type: Signature
          Rule: "OR('SpainMSP.admin', 'SpainMSP.peer', 'SpainMSP.client')"

        Writers:
          Type: Signature
          Rule: "OR('SpainMSP.admin', 'SpainMSP.client', 'SpainMSP.peer')"

        Admins:
          Type: Signature
          Rule: "OR('SpainMSP.admin')"

    AnchorPeers:
        - Host: hlf-peer--spain--peer0
          Port: 7051
```

Listing 4.4: Configtx Definition

network using PIVT. Though we have options for more advanced configurations, we deem it sufficient for a single cluster solution to use the current three files we have discussed. We discuss a multi-cluster solution in Chapter 5. The next subsection discusses how we exploit the Hyperledger Fabric network for controlling access to data providers' resources.

4.3.3 Interacting with Hyperledger Fabric and Chaincode

As mentioned in Section 2.2.5, Fabric supports chaincode written in Go, Java and Node.js. Furthermore, they provide different SDKs for invoking chaincodes. At the time of writing they provide SDKs for Node.js and Java, with plans of introducing SDKs for python and Go in future releases[13]. We consider using the unfinished python SDK due to our assumption of a linux-based operating system for data providers' resources. It follows from this assumption that the remote resource will have python installed, meaning no additional installations are required. However, the python SDK proves difficult to use due to the lack of proper documentation. Therefore, we choose the Node.js SDK for interacting with our chaincode, specifically, release 1.4. Refer to Appendix A for our Node.js scripts.

[13]https://hyperledger-fabric.readthedocs.io/en/release-1.4/fabric-sdks.html

Applications

When writing a Fabric application[14] for interacting with chaincode, there are six steps we must consider.

1. Export an identity from a *wallet*.
2. Connect to a *gateway*.
3. Access the desired network (channel).
4. Construct a *transaction request*.
5. Submit the transaction.
6. Retrieve the result.

The first step introduces the term *wallet*, which is a mechanism for holding one or several user identities. We utilize the SDK to create a *file system wallet*, and add identities to it. PIVT automatically enrolls one client and one admin per organization. We use the Node.js SDK to generate identities from the certificates and private keys of these users, and subsequently add them to our wallet.

After fetching an identity from the wallet, we must connect to a *gateway*[15], which will manage the network interactions on behalf of our application. To establish a connection, we use our identity along with a *connection profile*. The connection profile comprises information regarding the network we are connecting to. This may include addresses to peers, orderers and certificate authorities, as well as several connection options[16]. Refer to Appendix A for our connection profile.

After connecting to the gateway, we may access the desired channel with the `gateway.getNetwork(<channel_name>)` command. Note that this operation will only successfully return a channel instance if the identity used to connect to the gateway is a member of the channel.

From the channel instance, we may generate an instance of our chaincode using the `channel.getContract(<contract_name>)` command. This instance allows us to either *submit* transactions when we wish to add a new block to the ledger, or *evaluate* transactions when we simply wish to query a state. When submitting the transaction, we must

[14] https://hyperledger-fabric.readthedocs.io/en/release-1.4/developapps/application.html
[15] https://hyperledger-fabric.readthedocs.io/en/release-1.4/developapps/gateway.html
[16] https://hyperledger-fabric.readthedocs.io/en/release-1.4/developapps/connectionoptions.html

provide the name of the function we are invoking, as well as any parameters the function is expecting. If we are successful, the chaincode function will generate a payload and return it to us. The payload may be any arbitrary value presented as a stream of binary data. We may use this payload to return an OTC to data consumers.

4.3.4 Governing Data and Resources with Hyperledger Fabric

As previously mentioned, we may use channel policies to govern user privileges based on their roles. We can exploit this concept to decide who gets to perform computations on remote datasets, i.e., who gets to push Ansible commands on a data provider's resource. The participating organizations will agree which roles from which organizations will be granted privileges before the network is used.

As mentioned in 4.2, we exploit the fact that Ansible requires SSH access in order to govern access. Since we are assuming that the data provider's resource is running a Linux-based system, it would be sufficient for them to hold the data consumer's public key in their `authorized_keys` file. However, before the data provider can accept the public key of any alleged data consumer, they need to confirm that the consumer indeed is a member of the channel, and that their role is satisfactory according to the channel policies.

Our Chaincode

We introduce the chaincode `access-chaincode` to our channel. The chaincode is written in *Node.js*, and will contain functions for granting access to the resources of any participating data provider. As shown in Figure 4.1, our architecture proposes the use of an OTC for granting access. The `grantAccess` function will construct an OTC by combining the client ID of the person invoking the chaincode with the current datetime, and hash this value using the SHA-256 hash function. We assume the following to be true for cryptographic hash functions:

Collision resistance in cryptographic hash functions: *Given a hash function H, it should be infeasible to find two messages M and M', where $M \neq M'$, such that $H(M) = H(M')$.*[17]

This assumption ensures us that all OTCs are unique, and are safe for one-time usage. After generating the OTC, the chaincode function will commit the OTC to the ledger, and subsequently return it to the user. The function is shown in Listing 4.5.

[17] https://en.wikipedia.org/wiki/Collision_resistance

```
async grantAccess(stub, args) {
    const cid = new shim.ClientIdentity(stub);
    const provider = args[0];

    const dateTime = new Date().toLocaleString();
    const userId = cid.getID();
    const valToHash = dateTime.concat(userId);
    const digest = crypto.createHash("sha256").update(valToHash).digest("hex");

    await stub.putState(provider, Buffer.from(digest));

    const payload = Buffer.from(digest.toString());

    return payload;
}
```

Listing 4.5: Chaincode Function for Granting Access

From the `grantAccess` function we observe the `stub`[18], which contains the APIs between the chaincode implementation and the peers. From this, we can create a `ClientIdentity`, containing information about the user who invoked the chaincode. This is the ID we use in conjunction with the stringified datetime object to produce the OTC. We then hash this value, and commit it to the channel ledger. The `stub.putState()` function accepts a *key* and a *value* for adding state changes to the ledger. We use the provider's name as the key, which we expect to be the first argument of the `args` parameter. The value committed to the ledger must be a stream of binary data, which we achieve by transforming our hash digest with the Node.js `Buffer` module. Lastly, we return the hash digest, i.e. the OTC, as the payload.

A data consumer may now invoke this chaincode function to obtain an OTC. The OTC will be recorded on the ledger, where the data provider may query it to ensure the legitimacy of any alleged consumer. For this to be possible, we must add a function to our chaincode for querying the ledger. This function can be seen in Listing 4.6.

Observe the difference between invoking the `query` function as opposed to the `grantAccess` function. This function will not add a block to the ledger, since we are *getting* an existing state instead of *putting* a new state. After getting the OTC using `stub.getState()`, the function will return the OTC to whoever invoked it.

These functions may now be installed on the proper peers and instantiated on the channel. However, the data consumer and provider will need some logic for sending and handling access requests. We discuss this logic next.

[18]https://hyperledger.github.io/fabric-chaincode-node/release-1.4/api/fabric-shim.ChaincodeStub.html

```
async query(stub, args) {
      const provider = args[0];
      const otc = await stub.getState(provider);
      if (!otc || otc.length === 0) {
        throw new Error('OTC does not exist');
      }

      const payload = Buffer.from(otc.toString());

      return payload;
    }
```

Listing 4.6: Chaincode Function for Querying the Ledger

Sending and Handling Access Requests

In order for the data provider to verify that an alleged consumer is valid, they must ensure that the received OTC matches the OTC that is recorded on the ledger. The provider will then either accept or reject the consumer's public key based on the OTC comparison. First, we need a mechanism for transferring the OTC and public key from the consumer to the provider.

Firstly, the provider needs to listen for access requests. We write a *python* script which uses TCP sockets to listen for requests. We choose python due to the reason previously mentioned in Section 4.3.3. Upon receiving a public key and an OTC, the data provider's script will run our application for invoking the `query` function in our chaincode. This will return the last OTC committed to the ledger.

Next, the data provider's script will compare the OTC from the ledger to the one received from the alleged data consumer. If they are the same, the data provider will add the received public key to its `authorized_keys` file, thereby granting the data consumer passwordless SSH access, and return a message telling the consumer that access has been granted. However, if the OTCs do not match, the provider will discard the received public key and return a message telling the consumer that their OTC is invalid.

With this script running on the data provider, it is ready to receive access requests. Furthermore, the data consumer requires some logic such that they may connect to the provider's TCP socket, and send their obtained OTC and public key.

We write another python script for the data consumer's functionality. The script will use the address of the data provider along with the port on which they are listening for TCP connections. It will then run our application for invoking the `grantAccess` function in our chaincode, and subsequently send the obtained OTC and public key. The public key is obtained from a path, which the user will supply when running the script. Lastly, the script awaits a response informing the consumer that access has either been granted or denied.

When these operations are done, the data consumer will have passwordless SSH access to the data provider. Thus, we are ready to push Ansible commands and run computations. This is discussed in the next section.

4.4 Running Computations with Ansible Playbooks, Docker and Hadoop

In this section we discuss our process of developing Ansible playbooks for installing Docker, starting the necessary containers for a Hadoop cluster, and running a MapReduce job.

4.4.1 Docker Environment for Local Testing

Before implementing our solution, we set up an environment of one control node and two managed nodes, i.e. one data consumer and two data providers, to test our Ansible playbooks. We achieve this by setting up a Docker environment comprising one container per node. The *control container* will hold an installation of Ansible, and is the container from which we run our playbooks.

We use *docker compose* to automate the process of setting up the environment. Compose allows us to define and run a multi-container environment by defining a YAML configuration file. Here, we can specify all the services we want to run in an isolated environment.

Each container uses a *Dockerfile*[19], in which we can, e.g., specify what image the container should use and which ports it should expose, as well as copy files from the host system to the container. Furthermore, we use the `RUN` command to execute commands in a new layer on top of the current image. We use this feature to install Ansible and all its dependencies inside the control container, as well as create the necessary directories for SSH connections on both control and managed nodes.

4.4.2 SSH Connectivity

As mentioned in Section 2.3.2, we need SSH to be able to connect between the control node and managed nodes. This functionality does not come out-of-the-box with the

[19] https://docs.docker.com/engine/reference/builder/

Ubuntu 18.04 image, since Docker images are usually shipped with the bare minimum of software and services. Therefore, we use the `RUN` command inside each Dockerfile to install and start the SSH service, as well as create the necessary directories for storing public and private keys. These directories are crucial, since they will allow a passwordless SSH connection between our control node and managed nodes.

To facilitate the process of generating keys and storing them at the appropriate locations, we generate the keys on the host machine and copy them into the appropriate directories after they are created. An important note regarding Linux based systems is that the SSH and authorized keys directories must be created with proper permissions. If the SSH directories are too accessible, they will be ignored. The directories created and their permissions are shown in Listing 4.7 and Listing 4.8.

```
# Make directories for ssh and authorized keys
RUN mkdir root/.ssh
RUN chmod 700 root/.ssh
RUN touch root/.ssh/authorized_keys
RUN chmod 600 root/.ssh/authorized_keys
```

Listing 4.7: Managed node SSH setup

```
# Make directories for ssh key.
RUN mkdir root/.ssh
RUN chmod 700 root/.ssh

# Copy private and public ssh key into ~/.ssh/
COPY ssh/id_rsa root/.ssh
COPY ssh/id_rsa.pub root/.ssh

# Change permission to 600 for private key
RUN chmod 600 /root/.ssh/id_rsa
```

Listing 4.8: Control node SSH setup

4.4.3 Docker Compose

After defining our Dockerfiles, we use a *docker compose* file to automate the process of setting up our containers. In this file, we specify each container we want to start along with several configuration options. One of these options is to specify a Dockerfile we wish to build the container from. This option is called *build*, and we use it to point to our control and managed Dockerfiles.

Furthermore, we have the option to add *volumes*[20] to our containers. Volumes are a mechanism for persisting data which is generated and used by Docker containers. This entails that files from our host machine which are added to a volume are available within our containers. Moreover, changes made to these files are instantly reflected in our containers. This is useful regarding our Ansible playbooks and other scripts, since we will not have to restart our containers each time a change is made.

Since we are planning to install and run Docker containers inside of Docker containers, we need to add some options which grant the containers certain permissions. On Linux, Docker utilizes `iptables` to provide network isolation. However, the containers will not have permission to initialize certain tables by default. We solve this by adding the `NET_ADMIN` parameter to the *cap_add* option, and setting the *privileged* option to `true` in our docker compose file.

4.4.4 Installing Docker and Starting Hadoop Containers with Ansible

Our process contains four stages, with each stage defined in its own playbook.

- `install_docker`
- `pull_images`
- `start_containers`
- `run_job`

Docker Installation

We now have a Docker environment with three containers, which simulates one control node and two managed node. Next, we will use Ansible to install Docker and start Hadoop containers on the managed nodes. Both of our containers are running Ubuntu 18.04, and Docker provides several options for installing the Docker engine on an Ubuntu host[21]. The simplest way would be to use the convenience scripts provided by Docker, which is a shell script that will set up Docker engine for you. However, according to the documentation there is some risk involved with this, since it has to run with root or

[20] https://docs.docker.com/storage/volumes/
[21] https://docs.docker.com/engine/install/ubuntu/

sudo privileges, and it is therefore not recommended for all environments. We choose to install using the repository.

The *"Install using the repository"* option consists of installing packages to allow *apt* (a package manager used by Ubuntu) to use a repository over HTTPS. After the packages are installed, we can add Docker's official GPG (GNU Privacy Guard) key as well as their repository, and install the Docker engine. *DigitalOcean* already provides an Ansible playbook which performs these operations[22], as well as starts some containers. We may use this playbook, omitting the steps which pull images and start containers, for our purposes. We put these operations in the `install_docker` playbook.

Additionally, we observe that the Docker installation automatically uses the outdated *aufs* storage driver when installing and starting the Docker service. This proves to be problematic when starting our containers. For this reason, we add a *daemon.json* file, telling the remote Docker service to use the *vfs* storage driver. Refer to this link for more information on storage drivers in Docker.

Pulling Images and Starting Containers

After installing Docker we need to start the necessary containers for running a Hadoop job. First, we consult *Docker Hub*[23] for the Hadoop images we need. We observe that *Big Data Europe*[24] offers all the images needed for a Hadoop YARN architecture. They also include a `wordcount` job, and a `README.md` file to be used as an example job. Thus, we can use Ansible to pull these images from their repository to the remote resource. Ansible offers several modules for orchestrating Docker containers[25], amongst which we can use the `docker_image` module for pulling images. The images we can choose from are: namenode, datanode, resourcemanager, nodemanager and historyserver. For testing purposes, it is sufficient to pull the namenode and datanode images to launch a minimal cluster. These operations are defined in the `pull_images` playbook.

Furthermore, we need to start the containers with the proper configurations. Big Data Europe provides a Docker compose file in their GitHub repository, along with all their images. Running this file will build the images, start the containers and add them to a Docker network. However, if we were to use this to start the containers, the files would have to reside on the remote host. To avoid unnecessarily transmitting files to the remote host, we write an Ansible playbook which performs the necessary operations. We use the `docker_network` module to create a network for our containers,

[22]`https://github.com/do-community/ansible-playbooks`
[23]`https://hub.docker.com/`
[24]`https://www.big-data-europe.eu/`
[25]`https://docs.ansible.com/ansible/latest/scenario_guides/guide_docker.html`

and the `docker_container` module to create and start the containers with the proper configurations. These operations are defined in the `start_containers` playbook.

Running the Job

Furthermore, Big Data Europe provides a *Makefile* to perform the job, which will run the necessary commands inside the Hadoop base image. Again, using this file will require us to send it to the remote machine, which is undesirable. We instead write an Ansible playbook to run the necessary commands. Our playbook will create an `input` directory in HDFS, copy the attached `README.md` file into it, run the included `wordcount` job, place the output in the `output` directory in HDFS, and finally delete the input and output directories. Writing and testing complex jobs is outside the scope of this thesis, which is why we use the included `wordcount` job as a test computation, and the included `README.md` file as test data.

Since we want to retrieve these results before deleting the directories, we need to add a few lines of code. We simply copy the output directory from HDFS to the namenodes' local file systems, and then copy the directory from the namenode container to the managed nodes' local file system. These operations are defined in the `run_job` playbook.

When the results reside on the managed nodes' local file systems, we may perform the task of retrieving the results. We discuss this in the following section.

4.5 Private Data Collections for Retrieving Distributed Private Results in a Multi-Provider Scenario

The last problem we tackle is the task of collecting and combining the results from all data providers, while keeping data private between them. Simply putting the results in a state in the Hyperledger Fabric will expose the data to all participating peers and orderers. This may be undesirable in situations where data providers are reluctant towards sharing data. We propose using *private data collections* (PDCs) in Hyperledger Fabric for solving this problem.

4.5.1 Private Data Collections

When a transaction is committed to a channel ledger, it is broadcasted to all peers and orderers participating on the channel. As discussed in Section 2.2.5, every peer holds a physical copy of the channel ledger. This means that any organization may access all data that is transacted on the channel as long as they are a member.

We previously discussed how channels can be used to keep transactions transparent between a subset of organizations, while keeping them private from the rest of the network. However, consider the case where a subset of organizations on a channel needs to keep their transactions private. One possible solution would be to create separate channels for each of these cases, though, this would surely clutter the network, increasing complexity and introducing unnecessary configurations. Hyperledger Fabric introduced *private data collections*[26] to aid such cases.

When using a private data collection, there are two types of data being transmitted: The actual private data and a hash of the data. The actual data is only sent to peers from organizations who are authorized to see it, using a *gossip protocol*[27]. The private data is stored in a separate *private* database on the authorized peers, and is accessible from chaincode on these peers. The orderer is not involved in this process, keeping it private from orderer organizations as well. The hash of the data is treated as a normal transaction, meaning, is endorsed by peers, sent to the orderer for validation and broadcasted to every peer on the channel.

Note that we are required to set up anchor peers for this communication to work. We have briefly mentioned that anchor peers are used for cross-organization communication. This is essential when using private data collections, since the gossip protocol communicates private data peer-to-peer between authorized organizations.

Defining Private Data Collections

A private data collection is initialized when instantiating a chaincode by passing the `collections-config` parameter. This parameter must contain a path to a *collection definition*. The definition we use for our network is shown in Listing 4.9.

First, observe the `name` property, which is what a chaincode will use to identify the collection from which it will get data. The `policy` is what determines who may access the collections, in this case, any member of the specified organizations has access. However, one may define a more role-specific policy simply by replacing `member` with any role definition. The `requiredPeerCount` and `maxPeerCount` defines the minimum number of peers which must receive the private data, and the maximum number of peers which may receive the private data, respectively. Authorized peers which do not posess the

[26] https://hyperledger-fabric.readthedocs.io/en/release-1.4/private-data/private-data.html
[27] https://hyperledger-fabric.readthedocs.io/en/release-1.4/gossip.html

```
[
 {
    "name": "collectionNetherlandsStavanger",
    "policy": "OR('NetherlandsMSP.member', 'StavangerMSP.member')",
    "requiredPeerCount": 1,
    "maxPeerCount": 3,
    "blockToLive": 3,
    "memberOnlyRead": true
 },
 {
    "name": "collectionSpainStavanger",
    "policy": "OR('SpainMSP.member', 'StavangerMSP.member')",
    "requiredPeerCount": 1,
    "maxPeerCount": 3,
    "blockToLive": 3,
    "memberOnlyRead": true
 }
]
```

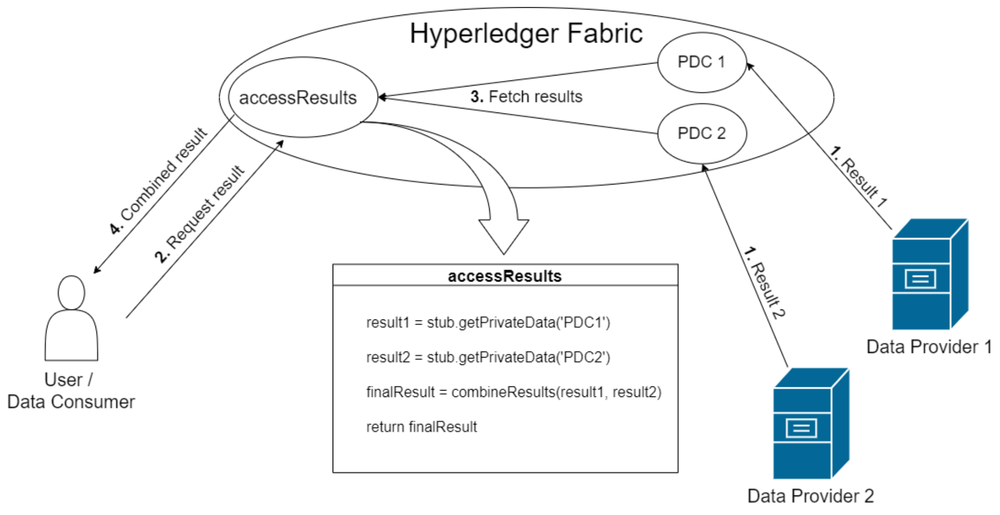
Listing 4.9: Collection Definition

private data may then pull it from other authorized peers, should they need to. These properties are not important in our test scenario, since our network comprises one peer per organization. Next, we observe the `blockToLive` parameter, which specifies how long data should be present in the private database, expressed in number of blocks. Once the number of blocks is reached, the private data is purged. In our definition, this means that for every third block committed to the ledger, the private data should be purged. Finally, the `memberOnlyRead` specifies that only members from the specified organizations may have read-access to the private data. One may set this value to `false` if one wishes to implement a more self-defined access control for different chaincodes.

4.5.2 Proposed Architecture

An illustration of our proposed architecture for using private data collections is shown in Figure 4.3.

In the figure, we observe 1) two data providers who will, upon completing their computations, put the results in their respective private data collection. The policies for these PDCs are shown in Table 4.2. With the results residing in their own collections, the data consumer may 2) invoke the `accessResults` chaincode, which is shown as pseudo-code in the figure. In short, the chaincode will 3) get each result from the private data collections, and then perform the necessary operations to combine the results. In our case, we have results from a wordcount job, meaning the function will transform the results to json



```
result1 = stub.getPrivateData('PDC1')
result2 = stub.getPrivateData('PDC2')
finalResult = combineResults(result1, result2)
return finalResult
```

Figure 4.3: Proposed Architecture for Retrieving Distributed Private Results

objects, sum values with matching keys, and add key-value pairs which are unique to an object. Refer to Appendix A for the entire `accessResults` chaincode. When the results are combined, the chaincode will 4) return the final json object (as a binary data stream).

Table 4.2: Policy for Each Private Data Collection

Collection	Policy
PDC 1	OR('Data Consumer', 'Data Provider 1')
PDC 2	OR('Data Consumer', 'Data Provider 2')

We mentioned that a private data collection is initialized when instantiating the chaincode. For this reason, we may not use PIVT for instantiating our chaincode, since it does not yet support private data collection definitions. Therefore, we use the Node.js SDK to instantiate the chaincode following the same steps for invoking chaincode, described in Section 4.3.3. We construct an instantiation request, which contains the `collections-config` parameter pointing to our collection definition, and send it as an instantiation proposal. If successful, we get a message telling us the chaincode has been instantiated.

After instantiating, we have a chaincode which may utilize the private data collections defined in our collection definition. The process illustrated in Figure 4.3 starts when computations are finished, and results have been produced. With PDCs in place, the last thing our Ansible playbooks will do is run a Node.js script which will invoke the `putPrivateCollection` chaincode. This chaincode is shown in Listing 4.10.

```
async putPrivateCollection(stub, args) {
    const result = args[0];
    const collection = args[1];

    await stub.putPrivateData(collection, 'result', Buffer.from(result));

    return Buffer.from(result.toString());
}
```

Listing 4.10: Chaincode Function for Putting Private Data into a PDC

From the chaincode, we observe a quite straight-forward approach, where the result and collection name are passed in as arguments. We then put the result in the specified collection under a `"result"` key, which the `accessResults` chaincode will use to fetch each result.

Once the Ansible playbook has finished, and the results are put in their respective collections, the data consumer may request the combined results. After the data consumer retrieves the combined result, we have completed the process. A data consumer has successfully gained access to multiple data providers, run computations on each of their data sets, and safely retrieved the combined result without sharing any data or results between data providers.

In the next chapter we will run three demonstrations using this system. We will run the network locally, in a single cluster on Microsoft Azure cloud, and in a distributed multi-cluster environment on Microsoft Azure.

Chapter 5

Deploying the System

5.1 Overview of Experiments

With the system in place, we will experiment with several ways of running the Hyperledger Fabric network. We will first describe how the system can be deployed locally, using a single node Kubernetes cluster in Minikube. Next, we will describe two ways of deploying the network in a cloud environment, namely, in Microsoft Azure. The sections will be structured as follows:

- Deploying the Hyperledger Fabric network locally with Minikube.
- Deploying the Hyperledger Fabric network in Azure using AKS.
- Deploying the Hyperledger Fabric network in a Distributed Cross-Cluster Environment in Azure using AKS.

5.2 Running the Hyperledger Fabric Locally with Minikube

To deploy the system, we need a running Kubernetes cluster in which we can deploy the Hyperledger Fabric network. Such a cluster can be run locally using a multitude of tools. On Windows or Mac, one can activate a Kubernetes cluster with a *Docker for Desktop* installation. However, when developing on Linux we must choose another option. We first consider *microk8s*[1], which is a lightweight Kubernetes installation running on a single node. However, this tool proves to be problematic when using Argo workflows. Therefore, we use Minikube[2] instead.

[1] https://microk8s.io/
[2] https://kubernetes.io/docs/setup/learning-environment/minikube/

```
spec:
  type: NodePort
  selector:
    name: hlf-peer
    app: hlf-peer--{{ $org.Name | lower }}--{{ $peer | lower }}
  ports:
  - protocol: TCP
    port: 7051
    name: grpc
    nodePort: 3000{{ $i | int }}
```

Listing 5.1: Extract from Peer Service Definition

Minikube is supported for Windows, Mac and Linux, since it runs a single-node cluster inside a VM on the host machine. After installation, one can simply launch a Kubernetes cluster by running `minikube start` in a terminal, however, we additionally specify the *driver*[3] which Minikube will use. We specify KVM as the driver (briefly mentioned in Section 2.1.1), since using the Docker driver presents interaction issues between our Docker containers and the Fabric network.

As mentioned in Section 4.3.2, Kubernetes leverages services for directing traffic to pods. In our case, these services are our peers, orderers and certificate authorities. We must expose these to outside resources, such that we may interact with the Hyperledger Fabric network, which may be done in several ways. When launching a Fabric network with PIVT's standard configurations, all services will be exposed through a *Cluster IP*. This address is internal, meaning it is only accessible to pods running inside the cluster. This is not sufficient for our purposes, since we must be able to access Fabric services from Docker containers running outside the cluster. Therefore, we introduce a *load balancer*[4].

A load balancer will distribute incoming network traffic to a set of backend resources. We may leverage this concept to allow outside communication into our Kubernetes cluster. Minikube supports several Kubernetes features, including *NodePorts*[5]. A NodePort will expose services at a static port on the Node's IP address, and will route traffic to the Cluster IP of the service. In our case, this means a service can be accessed using <MINIKUBE IP><NODEPORT>.

In order to configure our services to leverage NodePorts, we edit the `peer-service.yaml`, `orderer-service.yaml` and `ca-service.yaml` files, which are part of the PIVT project.

[3] https://minikube.sigs.k8s.io/docs/drivers/
[4] https://docs.microsoft.com/en-us/azure/load-balancer/load-balancer-overview
[5] https://kubernetes.io/docs/concepts/services-networking/service/

An extract from our peer service definition is shown in Listing 5.1. Observe the *type* of the service, this is where we specify that the peer service should be exposed through a NodePort. Furthermore, we specify the port number using the `nodePort` parameter under the `ports` definition. This definition ensures that port numbers are predictable, starting at 30000 and incrementing by one per peer we add.

Once all our services have been configured to use NodePorts, we may launch our network on the Minikube cluster. To interact with the network we simply update our connection profiles (introduced in Section 4.3.3) with the IP address of Minikube, and the port numbers of the NodePorts.

Realistically, one would not run a Hyperledger Fabric network locally. Ideally, one would have the network running on a set of servers, or use a *cloud provider*. In the next section, we will deploy our network in the cloud using Azure and AKS.

5.3 Deploying the Hyperledger Fabric Network in Azure using AKS

Azure[6] is a cloud service created by Microsoft. It offers a plethora of services, including the *Azure Kubernetes Service* (AKS). AKS offers a fully managed Kubernetes service, and allows users to easily scale their infrastructure when needed. In this section, we will utilize AKS for provisioning a multi-node Kubernetes cluster.

5.3.1 Setting up an AKS Cluster

Provisioning a Kubernetes cluster in Azure using the *Azure portal* is a simple affair. When in the portal, we first go to *Create a resource* and choose *Kubernetes Service*. Here, we may specify some basic settings, such as which subscription to use, a resource group, a cluster name and a region. We will later experiment with regions to create a multi-cluster distributed Hyperledger Fabric network.

After basic configurations, we specify the size of our *node pool*. The node pool will contain the nodes which will host the Hyperledger Fabric infrastructure. Here, it is essential that we first consider the number of nodes it takes to run our Hyperledger Fabric network. Our network comprises three peer organizations with one peer each, and one orderer organization. This results in three peers, three certificate organizations and one orderer; each requiring one node, i.e., we will need seven nodes in our node pool to launch the network.

[6]https://azure.microsoft.com/en-us/

Furthermore, AKS allows us to choose the size of each node. There are different specs for each choice, allowing the user to consider their need for number of CPUs, size of RAM, number of disks, etc. Note, we also consider that the peer node must contain at least two disks, since we need one disk for the peer, and one disk for holding copies of ledgers. It is sufficient for us to use the smallest VM size, which contains four disks.

In the *authentication* tab, we turn off *role-based access control* for simplicity's sake. This makes it easier for us to connect to the cluster. For *networking*, *integrations* and *tags*, we use the default values. We are now ready to review and create our cluster.

During setup, we observe that Azure restricts the number of CPUs one is allowed to have in a region. We originally use the free-trial subscription, however, during the reviewing phase of our cluster setup we receive an error saying our CPU quota has been exceeded. Therefore, we are forced to upgrade our subscription to a pay-as-you-go plan. This allows us to have up to ten CPUs running in each region, which is sufficient for our deployment.

Once the cluster has been successfully reviewed, we may create it and subsequently connect to it from the *Azure Cloud Shell*. This shell has the `kubectl` client pre-installed, which is the client used for interacting with a Kubernetes cluster. However, we must first configure `kubectl` to connect to our cluster. We do this with the `az aks get-credentials` command, and specify the name and resource group pertaining to our cluster.

Next, we must install the prerequisites for PIVT[7]. We first use the `wget` command to download all the binaries, and subsequently add them to our path. When all prerequisites are added, we may launch the network from the Azure Cloud Shell. Subsequently, we may create the channel, and install chaincode using PIVT's helm charts. However, when deploying our network in the cloud it is essential that we use a proper load balancer to activate external IP addresses for our services, i.e., we need external IP addresses for our peers, certificate authorities and orderers such that users may interact with them. Using PIVT, we may activate this behaviour by passing the `peer.externalService.enabled` and `orderer.externalService.enabled` flags, and setting them to `true`. This tells PIVT to include the *external services* definitions. An extract from the `peer-external-service.yaml` is shown in Listing 5.2.

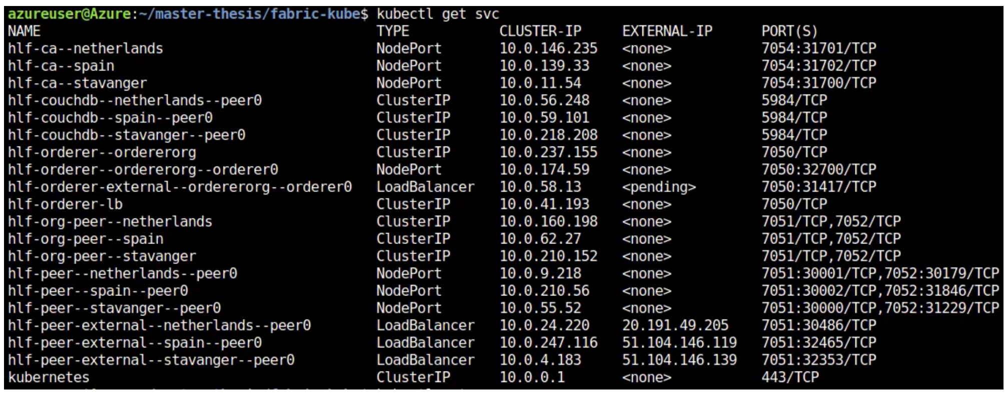
Observe the `type` field, where we specify the use of `LoadBalancer`. This will utilize the cloud provider's load balancer, as well as automatically create NodePorts and Cluster IPs which the load balancer will route traffic to. We may watch as the external IP addresses are being assigned. In Figure 5.1, observe the *External IP* column. Here, we see our external services obtaining IP addresses for external access. In the figure, we also observe that the status of the external orderer's IP is `pending`. This is because Azure is working

[7] https://github.com/APGGroeiFabriek/PIVT#requirements

```
spec:
  type: LoadBalancer
  selector:
    name: hlf-peer
    app: hlf-peer--{{ $org.Name | lower }}--{{ $peer | lower }}
  ports:
  - protocol: TCP
    port: 7051
    name: grpc
```

Listing 5.2: Extract from Peer External Service Definition

to assign a proper IP address, which may take a few minutes. Once all external services have received an IP address, we may access the network by updating our connection profile with the external IP addresses. e.g., we may access the Stavanger peer using 51.104.146.139:7051.



```
azureuser@Azure:~/master-thesis/fabric-kube$ kubectl get svc
NAME                                       TYPE           CLUSTER-IP     EXTERNAL-IP      PORT(S)
hlf-ca--netherlands                        NodePort       10.0.146.235   <none>           7054:31701/TCP
hlf-ca--spain                              NodePort       10.0.139.33    <none>           7054:31702/TCP
hlf-ca--stavanger                          NodePort       10.0.11.54     <none>           7054:31700/TCP
hlf-couchdb--netherlands--peer0            ClusterIP      10.0.56.248    <none>           5984/TCP
hlf-couchdb--spain--peer0                  ClusterIP      10.0.59.101    <none>           5984/TCP
hlf-couchdb--stavanger--peer0              ClusterIP      10.0.218.208   <none>           5984/TCP
hlf-orderer--ordererorg                    ClusterIP      10.0.237.155   <none>           7050/TCP
hlf-orderer--ordererorg--orderer0          NodePort       10.0.174.59    <none>           7050:32700/TCP
hlf-orderer-external--ordererorg--orderer0 LoadBalancer   10.0.58.13     <pending>        7050:31417/TCP
hlf-orderer-lb                             ClusterIP      10.0.41.193    <none>           7050/TCP
hlf-org-peer--netherlands                  ClusterIP      10.0.160.198   <none>           7051/TCP,7052/TCP
hlf-org-peer--spain                        ClusterIP      10.0.62.27     <none>           7051/TCP,7052/TCP
hlf-org-peer--stavanger                    ClusterIP      10.0.210.152   <none>           7051/TCP,7052/TCP
hlf-peer--netherlands--peer0               NodePort       10.0.9.218     <none>           7051:30001/TCP,7052:30179/TCP
hlf-peer--spain--peer0                     NodePort       10.0.210.56    <none>           7051:30002/TCP,7052:31846/TCP
hlf-peer--stavanger--peer0                 NodePort       10.0.55.52     <none>           7051:30000/TCP,7052:31229/TCP
hlf-peer-external--netherlands--peer0      LoadBalancer   10.0.24.220    20.191.49.205    7051:30486/TCP
hlf-peer-external--spain--peer0            LoadBalancer   10.0.247.116   51.104.146.119   7051:32465/TCP
hlf-peer-external--stavanger--peer0        LoadBalancer   10.0.4.183     51.104.146.139   7051:32353/TCP
kubernetes                                 ClusterIP      10.0.0.1       <none>           443/TCP
```

Figure 5.1: External IP Addresses for our Services

In Appendix A, we provide a video demonstration of the system in Azure. In the demonstration, we use the `README.md` file as our dummy data set and the included `wordcount.jar` file as our computational code. The same job is performed using the same data set on both data providers. Therefore, when using two data providers, we may conclude that the process is successful if we retrieve a result where the count of each word has doubled from its original result.

We have now deployed a functional Hyperledger Fabric network across several nodes in Azure using AKS. However, we are only utilizing one cluster. In a real-world scenario, organizations might want to host their infrastructure (peers, certificate authorities, etc.) in the cloud-provider of their choice or on their own premises. This would require multiple clusters, hosted in different parts of the world, communicating with each other to form a single Hyperledger Fabric network. Next, we will use PIVT to deploy our Hyperledger Fabric network on two AKS clusters, residing in different regions.

5.4 Deploying the Hyperledger Fabric Network in a Distributed Cross-Cluster Environment using AKS

In order for us to separate our Hyperledger Fabric infrastructure, we first need to create another cluster in AKS. We use the same configurations described in Section 5.3.1, except we now have to consider a different number of nodes. We divide our network as shown in Table 5.1.

Table 5.1: Overview of Clusters in Cross-Cluster Environment

Cluster	Region	Members	Nr. of nodes
1	North Europe	Spain and Netherlands	4
2	South-East Asia	Stavanger and Orderer	3

From the table, we observe that cluster 1 requires four nodes, while cluster 2 requires three. This is because a peer organization requires one node per peer, and one node per certificate authority, while an orderer organization only requires one node per orderer. Thus, we need four nodes for the two peer organizations in cluster 1, and three nodes for the peer and orderer organizations in cluster 2.

After we have created our clusters, we take inspiration from PIVT's *"Cross-cluster Raft network"* example[8]. Following this example, we first create two separate PIVT projects for each cluster by simply copying the files. Next, we alter the `network.yaml` and `crypto-config.yaml` files. In `crypto-config.yaml`, we have to specify *external peer organizations* for cluster 2, as well as an *external orderer organization* for cluster 1. Note that as opposed to PIVT's example, we do not enable TLS for our example. This is to simplify network communication for our proof-of-concept, however, it should be enabled in a production environment.

Another important difference when launching our cross-cluster network, is the use of *host aliases* and *external host aliases*. Host aliases are simply domain names along with their respective Cluster IP, while external host aliases are domain names along with their external IP. These are needed in order for the two clusters to be aware of each other's external services. To collect these host aliases, we first launch the network in a *broken state*, which means to launch the network without starting the peer and orderer pods. Before these are started, we will collect host aliases and external host aliases using shell scripts provided by PIVT. This needs to be done separately for each cluster. Furthermore, we need to copy the external host aliases of cluster 2 into the host aliases of cluster 1, and vice versa. Now, each cluster has the proper addresses for communicating with their external resources. Note that we are again using `LoadBalancer` for granting

[8] `https://github.com/APGGroeiFabriek/PIVT#cross-cluster-raft-network`

external IP addresses to our services. Therefore, it is important that we wait until all services have obtained an external IP before collecting external host aliases.

Afterwards, we may upgrade the network with the host aliases using a helm chart provided by PIVT. The setup of this cross-cluster example requires a number of operations performed on each cluster separately, and in the right order. To facilitate the process, and make it less error-prone, we write two shell scripts to automatically launch the network. Refer to Appendix A for our shell scripts. When each component is running on both clusters, we may create channels and install chaincode using the same helm charts as before, and subsequently instantiate the chaincode using our Node.js script.

Once these operations are done, we have a fully functioning multi-cluster Hyperledger Fabric network, with infrastructure residing in different parts of the world.

Chapter 6

Conclusion and Future Directions

6.1 Conclusion

In this thesis, we have presented an architecture for managing access to remote resources by utilizing chaincode in Hyperledger Fabric. Our system uses Ansible as a tool for provisioning computational resources in the form of Docker containers, which form a Hadoop cluster allowing us to run computations in an isolated environment on remote resources. Users who are enrolled in the Hyperledger Fabric network may invoke chaincode to obtain a one-time code for authentication, and subsequently send their public key along with the one-time code to gain passwordless SSH access to a data provider's resources. Furthermore, we have leveraged private data collections in Hyperledger Fabric for ensuring data privacy in a multi-provider scenario; allowing users to run computations on several data providers separately, and obtain a combined result. Finally, we have demonstrated how the system may be deployed using PIVT and Kubernetes running locally, and in Azure using AKS on a single cluster, as well as across two clusters residing in different regions of the world.

Our system allows organizations with common interests to collaborate without the need for complete trust. All activity is kept private from the rest of the network, while all data is kept private between the data providers on the channel.

6.2 Future Directions

In this section we highlight some caveats in our system and briefly discuss possible ways to solve them, as well as some added features to the system which would enhance its usability significantly. As previously mentioned, our implementation is a proof-of-concept, meaning we have taken some liberties in assuming certain scenarios, as well as simplified some implementations. These proposals are subjects to future work, which may be implemented to improve the system and progress it in the direction of a production-ready solution.

6.2.1 Securely Transporting the OTC and Public Key

As we mention in 4.2.1, there is a clear threat when communicating the OTC and public key as we do now. Our current implementation is simply sending the data through TCP sockets, which could prove devastating if malicious users intercept the traffic. They would then be able to replace the public key of the legitimate data consumer with their own, and gain access to remote resources. This is an example of a *man-in-the-middle* attack.

A common way of defending oneself against such an attack is using *TLS* (Transport Layer Security)[1]. TLS uses symmetric and asymmetric cryptography to provide secrecy and non-repudiation. Non-repudiation assures us that users cannot deny who they are, i.e., it provides assurances towards the origin of the data. A TLS implementation would drastically improve the system's tolerance against a man-in-the-middle attack.

6.2.2 Extending Computational Possibilities

As of now, the system uses a dummy computation along with a dummy dataset in order to demonstrate its functions. An interesting improvement would be to extend the computational possibilities of the system, such that it may handle more complex computations.

A relevant use for our system would be training machine learning models across several remote datasets. This would require a different approach to combining results or private datasets, possibly, a multi-party computation (MPC) protocol could be used to realize this functionality.

Chen et al. [24] investigate the possibilities of using SPDZ, a framework providing an MPC protocol, for machine learning algorithms such as linear regression and logisitic regression. Their work investigates the runtime and accuracy of SPDZ and conclude with promising results regarding the possibilities of extending SPDZ to more complex algorithms such as *neural networks*.

[1]https://www.thesslstore.com/blog/protecting-against-man-in-the-middle-attacks/

With a proper implementation for supporting machine learning, our system would be able to build machine learning models upon several distributed private datasets, without forcing the owners of the data to share it.

6.2.3 Integrating the Solution into TOTEM

An obvious future direction would be to integrate the solution with the rest of TOTEM's proposed architecture. Our system tackles the issues in TOTEM regarding data and resource governance, running computations at remote locations, as well as safely returning combined results in a multi-provider scenario.

Integrating our solution would entail implementing the customized computational framework, instead of using plain Hadoop. Furthermore, the computational code would have to be transacted on the blockchain. One way to solve this, would be to install and instantiate some chaincode which would evaluate the submitted computational code. If the code is deemed non-malicious, the chaincode will estimate a totem value, produce an OTC, and return them both to the user.

6.2.4 Removing Ourselves from Remote Resources

As the system stands today, we are not cleaning up after ourselves when computations are finished, meaning, we leave installations and Docker containers residing on the remote resource with no further purpose. Considering the case where the remote resources should be used for multiple purposes, and not just our system, it would be ideal for us to purge the system of our activity before exiting.

Such a purge would entail removing all containers and images, uninstalling Docker, and removing dependencies. However, some dependencies might have already existed on the remote resources, so we must be careful not to remove these. This can be facilitated with Ansible, which allows us to easily remove Docker containers and images, as well as uninstall dependencies. When we first connect and install our dependencies, Ansible will check whether or not the dependencies are there. We could harvest that information, and later use it for determining which dependencies we should uninstall, and which we should leave be.

Furthermore, we must remove the files and directories pertaining to our computations and results. Since we decide where these files and directories reside on the remote system, and therefore know exactly where they are, we may simply delete them using Ansible.

When these operations are completed, it is crucial that the SSH key of the data consumer is removed from the `authorized_keys` file on the remote resource. We wish that the user invokes the `grantAccess` chaincode function each time they run computations on a data provider's resource, which would not be necessary if the SSH key stayed in the data provider's `authorized_keys`. For this, one must add some functionality on the data provider's side, which would purge the current data consumer's public key upon a completed computation.

One must also consider the possibility of several data consumers being connected at once. In this case, it is essential that only remove the key pertaining to data consumers which have finished their computations is removed.

Lastly, in order for the OTC to be safe we must render it useless after one use. This could be achieved by the data provider submitting a state change to the blockchain. We could write chaincode which holds a state along with the OTC, which could either be `active` or `inactive`. With this approach, all data providers will be able to check if the OTC they received is still valid. If the status of the received OTC is `inactive`, the data provider will reject the access request and inform the requester. A data provider will set the OTC to `inactive` after it has added the data consumer's public key to its `authorized_keys` file.

List of Figures

List of Tables

Listings

Appendix A

Video Demonstration and Code

A.1 Video Demonstration

We provide a video demonstration of our system deployed on a single cluster in Azure using AKS.

Refer to this link for the video demonstration.

The video can also be found by search on YouTube, under the title: *"Demonstrating Secure Distributed Computing Managed by Hyperledger Fabric in Azure"*.

Explanations as to what is going on in the video along with time stamps are provided in the **video's description**, located under the video. Click on *"Show more"* for the full explanations.

A.2 Code

We provide a GitHub repository containing our code.

Refer to `https://github.com/jorgenholme/master-thesis` for the GitHub repository.

The repository is divided into three parts.

- **docker-files**: Contains the Docker compose file, Dockerfiles and all the necessary code to set up an environment with one data consumer and two data providers. The code for interacting with the HLF network is available under the `/fabric-scripts` located at the root directory of the control node. To open a bash shell into the control node, use the following command from the directory of the Docker compose file: `docker-compose exec control bash`.

Note, when launching the Hyperledger Fabric network for the first time, PIVT will automatically generate some users. These users are added to a local file system wallet, which we then use for accessing the network. To use our Docker containers, we must first create the wallet and then copy it into each Docker container.

- **fabric-kube**: Contains the PIVT project files needed to launch our network. Our network files can be found under `samples/ansible-nework/`, while the scripts for interacting with our network are found under `samples/ansible-nework-scripts/`. We also provide our `access-chaincode` under `samples/chaincode/access-chaincode/`.

  The scripts in `samples/ansible-network-scripts/javascript/addUsers/` may be used to add the already generated users to a local file system wallet. This wallet may then be used by other Node.js scripts to access the network.

- **cross-cluster**: Contains two PIVT projects for launching the network on two separate clusters. The first shell script for launching the network is found at `fabric-kube-orderer-stav/cross_cluster.sh`, run this from the directory it resides in. The second script is found at `fabric-kube-spain-netherlands/collect_host.sh`, run this from the directory it resides in. The second script must be run *after* all services on both clusters have received an external IP address.

Before performing any operations, install all prerequisites specified in PIVT's GitHub.

https://github.com/APGGroeiFabriek/PIVT

A.3 Launching the Network

To launch the network, use the following helm commands:

Create necessary crypto-config material, and prepare chaincodes:

./init.sh ./samples/ansible-network/ ./samples/chaincode/

Launch the network:

helm install ./hlf-kube –name hlf-kube -f samples/ansible-network/network.yaml -f samples/ansible-network/crypto-config.yaml

When all the pods are running (check with `kubectl get pod -watch`) create the channel:

helm template channel-flow/ -f samples/ansible-network/network.yaml -f samples/ansible-network/crypto-config.yaml | argo submit - –watch

Next, install the chaincode:

helm template chaincode-flow/ -f samples/ansible-network/network.yaml -f samples/ansible-network/crypto-config.yaml | argo submit - –watch

When chaincode is installed, we instantiate the chaincode using the `instantiate.js` script. This script may fail due to a timeout, but it will work after a few tries. This is due to some timeout restrictions set in Hyperledger Fabric.

When interacting with the Hyperledger Fabric network using our scripts, it is essential to first update the connection profiles with the correct IP addresses, i.e., update with the IP addresses and ports pertaining to the current network.

Bibliography

[1] D. T. Jose, A. Chakravorty, and C. Rong. Totem : Token for controlled computation: Integrating blockchain with big data. In *2019 10th International Conference on Computing, Communication and Networking Technologies (ICCCNT)*, pages 1–7, 2019.

[2] Clarify | Clarify Project site, . URL `http://www.clarify-project.eu/`. Library Catalog: www.clarify-project.eu.

[3] Jonas DeMuro December 18 and 2019. What is container technology? URL `https://www.techradar.com/news/what-is-container-technology`.

[4] What is a container? | app containerization | docker, . URL `https://www.docker.com/resources/what-container`.

[5] Overview of docker compose, . URL `https://docs.docker.com/compose/`.

[6] Michael Coleman. Containers are not VMs, 2016. URL `https://www.docker.com/blog/containers-are-not-vms/`.

[7] Peter Mell and Tim Grance. The NIST definition of cloud computing. Technical Report NIST Special Publication (SP) 800-145, National Institute of Standards and Technology, 2011. URL `https://csrc.nist.gov/publications/detail/sp/800-145/final`.

[8] Dylan Yaga, Peter Mell, Nik Roby, and Karen Scarfone. Blockchain technology overview. Technical Report NIST IR 8202, National Institute of Standards and Technology, Gaithersburg, MD, October 2018. URL `https://nvlpubs.nist.gov/nistpubs/ir/2018/NIST.IR.8202.pdf`.

[9] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, 2009. URL `http://www.bitcoin.org/bitcoin.pdf`.

[10] robin.materese@nist.gov. Blockchain, September 2019. URL `https://www.nist.gov/topics/blockchain`. Last Modified: 2020-01-16T20:09-05:00 Library Catalog: www.nist.gov.

[11] Nick Szabo. Smart contracts, 1994.

[12] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *white paper*, 3(37), 2014.

[13] Elli Androulaki, Artem Barger, Vita Bortnikov, Christian Cachin, Konstantinos Christidis, Angelo De Caro, David Enyeart, Christopher Ferris, Gennady Laventman, Yacov Manevich, et al. Hyperledger fabric: a distributed operating system for permissioned blockchains. In *Proceedings of the Thirteenth EuroSys Conference*, pages 1–15, 2018.

[14] A Blockchain Platform for the Enterprise — hyperledger-fabricdocs master documentation, . URL https://hyperledger-fabric.readthedocs.io/en/release-1.4/.

[15] Blockchain network — hyperledger-fabricdocs master documentation, . URL https://hyperledger-fabric.readthedocs.io/en/latest/network/network.html.

[16] Wo L Chang, Nancy Grady, et al. Nist big data interoperability framework: Volume 1, big data definitions. Technical report, 2015.

[17] HDFS architecture guide, . URL https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html.

[18] N. Z. Aitzhan and D. Svetinovic. Security and privacy in decentralized energy trading through multi-signatures, blockchain and anonymous messaging streams. *IEEE Transactions on Dependable and Secure Computing*, 15(5):840–852, 2018.

[19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[20] Jonathan Warren. Bitmessage: A peer-to-peer message authentication and delivery system. *white paper (27 November 2012), https://bitmessage. org/bitmessage. pdf*, 2012.

[21] Marcus Brandenburger, Christian Cachin, Rüdiger Kapitza, and Alessandro Sorniotti. Blockchain and trusted computing: Problems, pitfalls, and a solution for hyperledger fabric. *CoRR*, abs/1805.08541, 2018. URL http://arxiv.org/abs/1805.08541.

[22] F. Benhamouda, S. Halevi, and T. Halevi. Supporting private data on hyperledger fabric with secure multiparty computation. In *2018 IEEE International Conference on Cloud Engineering (IC2E)*, pages 357–363, 2018.

[23] APGGroeiFabriek/PIVT, May 2020. URL https://github.com/APGGroeiFabriek/PIVT. original-date: 2019-04-15T11:45:49Z.

[24] Valerie Chen, Valerio Pastro, and Mariana Raykova. Secure computation for machine learning with spdz. *arXiv preprint arXiv:1901.00329*, 2019.