UNIVERSIDADE DE LISBOA
FACULDADE DE CIÊNCIAS
DEPARTAMENTO DE INFORMÁTICA

Product Quality Certification Using Blockchain

Hugo Miguel Esteves Duarte

Mestrado em Engenharia Informática

Trabalho de Projeto orientado por:
Prof. Doutor Mário João Barata Calha

2024

Acknowledgments

I want to express my deepest gratitude to Professor Mário Calha for his invaluable guidance and patience throughout the development of this report. His insightful advice and unwavering support were crucial to completing this work.

I also want to extend my heartfelt thanks to the entire team at Sensefinity for warmly welcoming me and providing immense support during this journey. The knowledge and skills I have gained through this experience are invaluable, and I am truly grateful for the opportunity to learn and grow alongside such dedicated professionals.

A special thank you goes to my family for their unwavering support and belief in me. Their constant encouragement and faith in my abilities have been instrumental in my academic journey. I am here today, completing my master's degree, because of their love and support.

This work was financially supported by Project Blockchain.PT – Decentralize Portugal with Blockchain Agenda, (Project no 51), WP 1: Agriculture and Agri-food, Call no 02/C05-i01.01/2022, funded by the Portuguese Recovery and Resilience Program (PPR), The Portuguese Republic and The European Union (EU) under the framework of Next Generation EU Program.

To my Family.

Resumo

Este relatório apresenta uma solução arquitetónica abrangente para a integração de um sistema de cadeia de blocos na gestão da cadeia de suprimentos da Cereja do Fundão, utilizando o Hyperledger Fabric. A solução proposta visa construir uma rede descentralizada que envolve todos os atores-chave da cadeia: agricultores, centros de controle de qualidade, armazéns, transportadores e retalhistas. Este sistema atende a vários requisitos funcionais e não funcionais, como escalabilidade, segurança, privacidade e eficiência de custos. O desenvolvimento desta solução envolveu uma análise detalhada da arquitetura ideal, sem considerar os custos, bem como ajustes necessários para a implementação de uma fase piloto mais acessível, equilibrando funcionalidade e custos.

A arquitetura ideal propõe um sistema totalmente descentralizado. Neste modelo, cada organização dentro do consórcio é responsável por gerir de forma independente os seus próprios pares, ordenadores e autoridades certificadoras. Este design é baseado nas melhores práticas e recomendações do Hyperledger Fabric, que sugerem a separação das funções de aplicação e de ordenação entre diferentes organizações para garantir uma maior resiliência e evitar a centralização de componentes críticos. Ao evitar que uma única organização controle todos os componentes críticos, esta abordagem visa maximizar a resiliência do sistema e garantir a adesão aos princípios descentralizados fundamentais.

Cada organização participa com pelo menos um par e um ordenador. Os pares são responsáveis por manter o estado da cadeia de blocos e executar o código de cadeia, enquanto os ordenadores são responsáveis pela ordenação das transações e pela manutenção da integridade do livro-razão. Cada organização também deve operar com uma autoridade certificadora dedicada para gerenciar identidades e certificados TLS, garantindo uma comunicação segura entre os nós e um controle de acesso eficiente dentro da rede. Essa configuração ideal assegura alta disponibilidade e tolerância a falhas, alinhando-se aos requisitos não funcionais estabelecidos.

No entanto, a implementação prática do sistema foi ajustada para uma fase piloto devido às restrições financeiras. Esta fase simplificada da arquitetura reduz o número de pares e ordenadores e utiliza uma única autoridade certificadora por organização, equilibrando as necessidades de funcionalidade com a limitação de custos. O design para a fase piloto ainda é capaz de atender aos requisitos essenciais do projeto, estabelecendo a infraestrutura necessária para a gestão de transações, identidades e operações de rede, mas de uma maneira mais econômica. As mudanças foram projetadas para reduzir os custos de desenvolvimento e configuração, sem comprometer a funcionalidade fundamental do sistema.

A arquitetura da rede de cadeia de blocos é estruturada em canais distintos para garantir a privacidade e a segurança dos dados entre os membros do consórcio. Cada canal é responsável por um tipo específico de informação, como dados sobre a qualidade das cerejas, informações de transporte e contratos de pagamento. Esta abordagem em canais permite que as organizações compartilhem e armazenem dados sensíveis de maneira segura e controlada. Cada canal é gerido por cadeias de código dedicadas, que facilitam operações CRUD (criação, leitura, atualização e exclusão) relacionadas aos ativos dentro do canal. A implementação de chaincodes separados por tipo de ativo segue a prática recomendada de separação de preocupações, promovendo modularidade e facilitando a manutenção e a segurança.

O consenso Raft foi escolhido para a ordenação das transações na rede de cadeia de blocos. O Raft é recomendado para a versão 2.5 do Hyperledger Fabric e oferece uma solução de consenso eficiente, que garante a consistência e a integridade das transações. O algoritmo Raft permite que todos os nós ordenadores mantenham uma sequência consistente de transações, atendendo ao requisito de consistência não funcional. Além disso, o CouchDB foi selecionado como o banco de dados de estado dos pares devido às suas funcionalidades avançadas, como armazenamento em formato JSON, suporte a índices e consultas complexas, o que é crucial para a gestão eficaz dos dados.

A arquitetura de implementação utiliza o Kubernetes para containerizar todos os componentes da rede de cadeia de blocos, permitindo uma gestão eficiente dos recursos e a escalabilidade da solução. No Kubernetes, cada componente, como pares e ordenadores, é executado em pods dedicados, e namespaces são utilizados para separar e isolar recursos entre diferentes organizações. O Kubernetes facilita a autoescalabilidade através do Vertical Pod Autoscaler (VPA), que ajusta dinamicamente os recursos alocados para cada pod com base na demanda. Adicionalmente, o Cluster Autoscaler do Google Cloud é utilizado para ajustar o número de nós no cluster, garantindo que haja recursos suficientes para os pods e otimizando os custos associados.

A utilização do Kubernetes Gateway API para expor os serviços fora do cluster proporciona vantagens significativas em termos de flexibilidade e eficiência de custo. O Gateway API facilita o roteamento de tráfego de forma mais flexível e simplificada, direcionando o tráfego para os serviços apropriados através de recursos como HTTPRoute. Além disso, a configuração do Gateway API permite uma distribuição eficiente de tráfego entre múltiplas réplicas de um serviço, garantindo um gerenciamento otimizado dos serviços expostos.

Para garantir a persistência dos dados, a arquitetura utiliza Persistent Volumes (PVs) ligados a um Network File System (NFS). O NFS fornece um armazenamento centralizado, mantendo os dados da cadeia de blocos mesmo após reinicializações ou falhas dos pods. Os PVs são mapeados para diretórios específicos no servidor NFS, garantindo a separação e o controle de acesso entre as organizações. Persistent Volume Claims (PVCs) são usados para abstrair o acesso aos PVs, facilitando a gestão de dados e a operação contínua da rede.

Além disso, a cadeia de código é implementada como um serviço externo dentro do cluster Kubernetes, o que oferece benefícios significativos em termos de modularidade e simplicidade.

Em vez de instalar a cadeia de código em cada par, ela opera independentemente, o que simplifica a atualização e reduz a carga sobre os pares. Essa abordagem é especialmente útil em redes maiores onde a gestão e a manutenção da cadeia de código podem ser complexas.

Adicionalmente, foi realizada uma métrica para avaliar o desempenho do sistema utilizando a ferramenta Hyperledger Caliper. O Caliper é integrado ao Hyperledger Fabric e mede o desempenho da rede através de indicadores como taxa de transferência, taxa de envio, taxas de sucesso e falha de transações, latência e utilização média de recursos. Foram conduzidos três tipos de testes para leitura e criação de ativos, variando o número de máquinas e configurações de CPU e memória.

Os testes foram realizados em um ambiente de preparação dentro de um cluster Kubernetes no Google Cloud, utilizando máquinas da série E2. Os resultados mostraram que a taxa de transferência e a latência variaram de acordo com o número de trabalhadores e as taxas de transação, com o desempenho ideal alcançado em configurações específicas. Observou-se que a latência diminuía com volumes de transação maiores, possivelmente devido a mecanismos de concorrência do Hyperledger Fabric. No entanto, em taxas de TPS mais elevadas, ocorreram falhas de transação, destacando a importância de otimizar essas taxas por meio de ajustes no número de trabalhadores, aumento de recursos ou outras configurações no Hyperledger Fabric para garantir a escalabilidade e eficiência do sistema em ambientes de produção.

Esses testes demonstraram que, com a aplicação de configurações otimizadas, a arquitetura proposta é capaz de atender aos requisitos não funcionais de eficiência, escalabilidade e baixos tempos de latência, tornando o sistema robusto e preparado para operar em diferentes cenários.

Em resumo, este relatório apresenta uma solução arquitetónica robusta e escalável para a gestão da cadeia de suprimentos da Cereja do Fundão, baseada em cadeia de blocos. O protótipo atende tanto aos requisitos funcionais quanto aos não funcionais, demonstrando resultados promissores nos testes iniciais. Está atualmente em fase de validação real para confirmar a sua prontidão para produção. Embora algumas funcionalidades avançadas ainda estejam em desenvolvimento, o sistema revela um grande potencial para melhorar significativamente a transparência e eficiência da cadeia de suprimentos. A solução oferece um equilíbrio eficaz entre funcionalidade e custo, preparando o terreno para uma implementação detalhada e avaliação futura desta arquitetura inovadora.

Palavras-chave: Rastreabilidade Alimentar, Cadeia de Abastecimento, Blockchain, Hyperledger Fabric, Arquitetura de Sistema

Abstract

This report presents a blockchain-based traceability system for the Cereja do Fundão supply chain, developed with Hyperledger Fabric to enhance transparency, security, and accountability through a decentralized network involving farmers, quality control centers, warehouses, transporters, and retailers. The system addresses both functional and non-functional requirements, including scalability, privacy, and cost-efficiency. Initially envisioned as a fully decentralized system, the pilot phase adapted to financial constraints by simplifying the architecture—reducing peers and orderers, and utilizing a single Certificate Authority (CA) server per organization.

Deployment is managed using Kubernetes, which containers all components and leverages auto-scaling to optimize resource allocation. This setup facilitates dynamic adjustments of resources, ensuring efficient performance and cost-effectiveness. The system employs distinct channels for managing various data types, such as quality metrics, transportation details, and payment contracts, to ensure privacy and security. Each channel is managed by dedicated chaincodes, enhancing modularity and ease of maintenance. The Raft consensus algorithm ensures transaction consistency and integrity, while CouchDB supports advanced data management with efficient querying capabilities.

Performance benchmarks using Hyperledger Caliper were conducted across different machine configurations, assessing the blockchain system's throughput, latency, and resource utilization under varying numbers of workers and transaction per second (TPS) rates. The findings led to the successful application of optimal configurations, which improved system performance and reliability in the pilot deployment.

The prototype meets core functional requirements and has shown promising results in initial tests. It is currently undergoing further real-world validation to confirm its readiness for production. While some advanced features are still in development, the system demonstrates significant potential for enhancing supply chain transparency and efficiency, offering a robust and scalable framework for future industry applications.

Keywords: Food Traceability, Supply Chain, Blockchain, Hyperledger Fabric, System Architecture

Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

The prevailing global food system is currently facing a breakdown. It prioritizes quantity and short-term efficiency over taste, sustenance, and quality, creating a distance between consumers and the origins of their food. This separation has been connected to health problems like diabetes, cancer, and obesity, which are becoming more common. The excessive food miles, alongside industrialization, have led to food contamination, including salmonella poisoning, Mad Cow disease, and E. coli outbreaks. Moreover, consumers are increasingly disconnected from the physical, social, and intellectual origins of their food, primarily due to the prevalence of a cheap food system that emphasizes quantity and convenience over nutritional value and environmental sustainability. All these problems above have contributed to a shift in the consumer mind, opting toward a more sustainable and healthy option in the market [7].

The EU Digital Product Passport is a new concept derived from the Sustainable Product Regulation proposed by the European Commission on March 30, 2022, as part of the EU Green Deal [20]. This regulation establishes rights and duties for all stakeholders, emphasizing the role of traceability in all sectors, exempting food and medicine. The proposal mandates efficient solutions to enhance accountability and transparency across the supply chain. By addressing the specific challenges and requirements of the food sector industry, a dedicated traceability system can further improve food safety, supply chain efficiency, and consumer confidence. The exclusion of the food sector from the product passport offers this unique opportunity to implement this initiative within this sector. Furthermore, Gartner predicts that by 2025, 20% of the top global grocers are expected to adopt blockchain food safety and traceability to create visibility to production, quality, and freshness [17]. This aligns with the goals of the EU regulation, emphasizing the importance of traceability in the supply chain for sectors like food.

In today's supply chain management, building and preserving trust and reliability are crucial tasks. This is especially true in the complex world of modern supply chains, particularly within agriculture. With a wide array of players involved, from farmers to distributors to consumers, the accuracy of every transaction and record is vital. Ensuring that stored records are secure and unchangeable is a basic necessity. If these records were tampered with, it could lead to serious consequences such as legal issues, damage to reputation, and even risks to consumer safety. Blockchain technology offers a tailored solution to address these concerns [46]. Its key features, like im-

mutability and decentralization, mean that once data is entered, it can't be tampered with and is only accessible to authorized users. The decentralized nature of blockchain removes the need for a central authority or middleman, making transactions smoother and increasing transparency. Each transaction is checked and recorded by a network of users, ensuring everyone agrees and reducing the chance of fraud. This transparency and ability to verify information build trust among participants and provide a reliable record of events throughout the supply chain. Real-world examples demonstrate how blockchain technology enhances these capabilities. For instance, Walmart has integrated blockchain via IBM Food Trust to track products like pork and mango, improving transparency across their supply chain [35]. Similarly, Nestlé has employed IBM Food Trust to enhance traceability for their Zoégas coffee brand, providing detailed insights into the product's origin and journey [42]. These applications showcase blockchain's effectiveness in ensuring transparency and traceability within supply chain operations. These unique features make blockchain the preferred option for securing supply chain processes, distinguishing it from alternative solutions like distributed databases [53].

This project aims to establish a robust decentralized traceability system, specifically focusing on cherries sourced from the renowned Fundão region within the Cereja do Fundão supply chain, as part of the PRR project in Portugal to decentralize agricultural and supply chain processes [3]. Cherries originating from Fundão are widely recognized for their exceptional quality and distinct characteristics, thus distinguishing them significantly in the market landscape. Consequently, there is a pressing need to implement a monitoring system that meticulously traces their journey from origin to the ultimate consumer, thereby ensuring the integrity of their provenance. The significance of this initiative lies in its capacity to uphold the integrity and credibility of the Cereja do Fundão brand through the establishment of transparent and reliable traceability mechanisms. Leveraging the capabilities of blockchain technology, the project aims to develop an immutable ledger that comprehensively records every stage of the cherry supply chain. This approach not only fosters greater consumer confidence but also serves to safeguard the esteemed reputation of this product within the market milieu. This collaborative endeavor encompasses cherry producers from the Fundão region, the quality control center Cerfundão, Logistom responsible for transport logistics, distribution warehouses operated by the STEF group across the country, and the final retailer, Sonae. By collectively forming a comprehensive supply chain network, the project seeks to extend its impact beyond cherries to encompass other facets of the agricultural sector potentially.

To establish comprehensive traceability within the Cereja do Fundão supply chain, packaged cherries will be equipped with advanced sensors provided by Sensefinity [48], a prominent company specializing in IoT hardware and software solutions, and the institution where this project report is conducted. These sensors enable the collection of crucial data in near real-time, including GPS coordinates, temperature, humidity, and luminosity. By embedding these sensors in both the field crates and packaging labels, a heightened level of monitoring is facilitated, ensuring enhanced visibility into the conditions during transportation and storage. This deployment of sensor technology not only guarantees data accuracy but also reduces the risk of human errors in the

recorded information. The integration of these sensor-generated data into the blockchain serves as a pivotal mechanism to bolster accountability throughout the supply chain process, attributed to the immutability factor inherent in blockchain technology. Furthermore, while each entity within the consortium currently manages its system and databases for recording cherry-related data, the integration of blockchain with these existing systems will necessitate the development of APIs and the adaptation of interfaces. This strategic approach ensures that stakeholders can achieve transparent data sharing and uphold interoperability seamlessly. The integration with existing systems and the recording of sensor data in the blockchain are future tasks that will follow this project, as they require the foundational blockchain infrastructure being built in this project.

The system is constructed using the Hyperledger Fabric framework for permissioned blockchains [4], an emerging open-source technology widely employed in various proposed supply chain management systems (see Related Work). Permissioned blockchains are typically employed in scenarios where access to the blockchain network is restricted to a defined group of participants, and data viewing is limited to authorized entities [21]. This aspect is particularly crucial as not all data gathered from every step in the supply chain can be visible to every entity within the blockchain. Additionally, Hyperledger Caliper [22], part of the Hyperledger project, is utilized to conduct benchmark tests on the blockchain, ensuring alignment with the project requirements identified by the interoperability team. Furthermore, the deployment of this system takes place in a cloud environment, specifically within a Kubernetes cluster on Google Cloud. This deployment choice is made to enhance availability and leverage the benefits offered by Kubernetes in deployments. Additionally, an application is developed to visualize data within the blockchain, providing stakeholders with a user-friendly interface to interact with the system and access relevant information.

Creating permissioned blockchains poses challenges due to their early-stage development. The limited number of available options and lack of maturity make the implementation process complex. Additionally, the absence of support from cloud providers for blockchain solutions necessitates the adoption of a cloud-agnostic approach, adding further complexity to the deployment process. Furthermore, achieving decentralization requires additional infrastructure, thus leading to increased expenditure. This can pose a significant challenge, especially when operating with a limited budget.

1.1 Goals

The objective of this project is to develop the initial version of an innovative system aimed at ensuring traceability of Cereja do Fundão cherries from origin to consumer, with a specific focus on the blockchain component. This involves creating the necessary infrastructure for each organization within the consortium to establish a fully decentralized system.

Once the system is operational, members of these organizations will need to record and access cherry-related information from the blockchain, with different members having varying permissions for creating and reading data. This includes ensuring that certain organizations only have access to specific subsets of data related to an asset, based on the permissions granted to the

individual accessing the information, thereby ensuring confidentiality and data protection. This necessitates the development of an API for interacting with the smart contracts that manage these assets within the blockchain, as well as an application for visualizing this data.

The data inserted into the blockchain must be accurate. This requires integration with the existing systems of the organizations and incorporation of sensor data throughout the process. However, this integration is planned for future phases of the project, pending validation of the foundational blockchain being constructed.

Taking these factors into account, the following main objectives are identified:

1. Develop the blockchain infrastructure for every organization within the consortium.
2. Implement mechanisms to securely record and access data on the blockchain:
   (a) Build an API for interacting with blockchain-managed smart contracts.
   (b) Develop an application for visualizing cherry-related data sourced from the blockchain.
3. Implement access control mechanisms on the blockchain to ensure data confidentiality and security.
4. Ensure traceability from cherry origin to consumer using blockchain technology:
   (a) Integrate blockchain data with existing organizational systems.
   (b) Incorporate sensor data throughout the traceability process to enhance accuracy and transparency.

1.2 Contributions

The contributions of this work encompass:

1. The prototype of a comprehensive blockchain system to advance supply chain technology.
2. Architectural improvements for the existing infrastructure of Sensefinity: new microservices, data models, and REST API expansion.

1.3 Structure of the document

This document is organized as follows:

- Chapter 2 - Provides a comprehensive overview of the key technologies foundational to the proposed traceability system;
- Chapter 3- Reviews studies on blockchain frameworks, consensus algorithms, state databases, and product traceability.

- Chapter 4 - Identifies the functional and non-functional requirements by examining the roles and processes within the Cerejas do Fundão supply chain.
- Chapter 5 - Provides an overview of the system architecture, detailing infrastructure, network, and Kubernetes components.
- Chapter 6 - Presents the blockchain implementation using Hyperledger Fabric framework;
- Chapter 7 - Describes the smart contract and API implementation for data insertion into the application.
- Chapter 8 - Assesses the performance and cost implications of the blockchain implementation using the Hyperledger Caliper benchmarking tool.
- Chapter 9 - Concludes the work and presents directions for future research.

Chapter 2

Background

The background section provides a comprehensive overview of the key technologies shaping the proposed traceability system. The chapter is divided into:

- **Section 2.1**: An exploration of blockchain technology, covering its foundational concepts and various types.
- **Section 2.2**: Hyperledger Fabric components are explained, including the transaction flow execute-order-validated.
- **Section 2.3**: Provides an overview of cloud service models and details the specific model used in the project.
- **Section 2.4**: Introduces Kubernetes, explaining its role as an open-source container orchestration system.
- **Section 2.5**: Covers Hyperledger Caliper, a benchmarking tool used to measure the performance of blockchain networks.
- **Section 7.4**: Provides a summary of this chapter.

2.1 Blockchain

The origins of blockchain lie in a paper published in 2008 by a person known as Satoshi Nakamoto [41], outlining the framework for Bitcoin. As technology pundits and gurus recognized new use cases for this technology beyond cryptocurrency, the term "blockchain" was formalized and therefore, is a designation absent in Satoshi's original paper.

Blockchain is a peer-to-peer, append-only, transaction log. Append-only means data once written cannot be erased, in other words, it's immutable. Peer-to-peer or decentralized, allows network participants to conduct transactions without the need for a third-party intermediary. A transaction log essentially serves as an electronic record. Transactions are secured by using digital signatures and blockchains usually have a feature known as a smart contract. A smart contract is

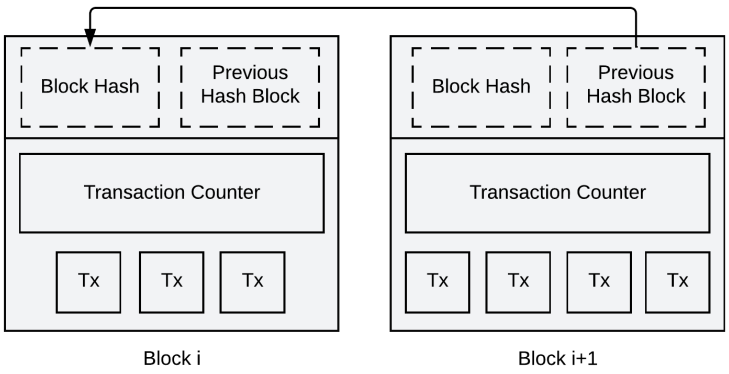

Figure 2.1: Blockchain Architecture

a piece of code that can respond to user input and generate a transaction response. Blockchain stores transactions in a data structure called blocks, as illustrated in Figure 2.1.

Each block of the blockchain contains a header and a body. The header includes the essential information of the block and the hash of the previous one, forming a chain that is cryptographically linked together. The body contains a transaction counter and the actual transactions. The maximum number of transactions in a block depends on factors such as block size and the size of each transaction [56]. Before a transaction is added to the blockchain, it undergoes a validation and consensus process that differs in each type of blockchain - permissionless and permissioned [21].

2.1.1 Permissionless Blockchains

Permissionless blockchains, such as Bitcoin and Ethereum [10], are open to anyone who wants to participate. In these types of blockchains, there is no central authority controlling the network. Anyone can join the network, participate in the consensus process, and add new blocks to the chain. The validation of transactions and blocks is typically done through mechanisms like Proof of Work (PoW) or Proof of Stake (PoS), where participants (miners or validators) compete or collaborate to solve cryptographic puzzles or stake their assets as collateral. This decentralized nature enhances security and transparency but often comes at the cost of higher energy consumption and slower transaction speeds due to the complexity of the consensus mechanisms.

2.1.2 Permissioned Blockchains

Permissioned blockchains, on the other hand, are closed networks where only authorized participants can join and participate in the consensus process. These blockchains are often used by businesses and organizations that require more control over who can read, write, and validate transactions. Hyperledger Fabric [4] and R3 Corda [9] are examples of permissioned blockchains. In these networks, consensus algorithms are more streamlined and efficient since they do not need to cater to an open, anonymous environment. Common consensus algorithms used in permissioned blockchains include Practical Byzantine Fault Tolerance (PBFT) and its variations [11].

PBFT is designed to provide high fault tolerance and efficiency in a closed network by allowing a group of pre-approved nodes to agree on the state of the blockchain. This results in faster transaction processing and lower energy consumption [47]. However, the trade-off is that permissioned blockchains are less decentralized and rely more on trust among the participants. In Hyperledger Fabric, for example, the consensus process is managed through identity and policy mechanisms. At the same time, transaction access control is implemented through channels and private data collections, as explained in the next section.

2.2 Hyperledger Fabric

Hyperledger Fabric, hosted by the Linux Foundation, is an open-source blockchain platform, which falls within the category of permissioned blockchains [4]. Fabric has three distinct types of nodes: peers, orderers, and clients. Additionally, the platform adopts multiple mechanisms to implement access control. The different nodes along with their respective components are described as follows:

1. **Peers**: Peers host the ledgers and smart contracts(when packaged it is called chaincode). In addition, peers participate in transaction approval, as endorsers, and they are responsible for updating the ledger, as commiters.

   (a) **Ledger**: The ledger has two components, namely, the world state and blockchain. The world state is a database that stores data in the form of <key, value, version>, where the last parameter is used for concurrency control, increasing every time the key value is updated. The blockchain stores all transactions that result in the current world state.

   (b) **Chaincode**: Chaincode is a list of smart contracts that were packaged to be deployed in the network. These smart contracts contain the business logic which may perform read, write, or update operations on the world state. Before a chaincode is committed to the peer, all organizations specified in the endorsement policy have to agree on the configuration of the chaincode. Likewise, transactions need to be signed by the endorsers specified before they are considered valid or invalid.

2. **Orderers**: Every network has at least one ordering node responsible for executing a consensus mechanism, being Raft the officially supported, that establishes the order of the transactions that were endorsed. Succeding this process, the ordering service bundles the transactions into blocks and sends those to the endorsing peers specified in the endorsement policy so the ledger is updated.

3. **Clients**: Clients can use an application that executes chaincode functions hosted in peers through a gRPC gateway. The operations take effect only after the transaction is validated as valid by all peers. Optionally, the application can receive the ledger update event, if programmed for that.

4. **Access Control Mechanisms**: Every node has an identity and policies are used to manage access to particular resources or make modifications to the system. Furthermore, channels are a mechanism for communication and data confidentiality.

   (a) **Identities**: In Hyperledger Fabric, each participant is assigned a digital certificate as proof of identity. These certificates are issued by a Membership Service Provider (MSP) and are bonded to specific organizations within the consortium. This identity management system ensures that each participant's actions can be authenticated and traced back to their organization.

   (b) **Policies**: Policies in Hyperledger Fabric are logical expressions used to control and evaluate whether specific tasks are performed by the required identities. They are divided into two types: signature policies and implicitMeta policies.

      i. **Signature Policies**: These policies utilize logical operators such as AND, OR, and NOutOf to determine the required endorsements for a transaction. For example, the expression "AND(Org1.peer, Org2.peer)" indicates that both identities from Org1 and Org2 must sign the transaction for it to be considered valid under this endorsement policy.

      ii. **implicitMeta Policies**: These policies aggregate the outcomes of multiple signature policies using logical operators like ALL, MAJORITY, and ANY. For instance, the expression "MAJORITY Endorsement" means that a majority of the signature policies labeled as "Endorsement" must return true for the overall policy to be satisfied. In essence, while signature policies are defined for individual entities, implicitMeta policies combine these outcomes to determine the collective endorsement decision.

   (c) **Channels**: Channels act as private networks between organizations. Their functionality is to keep transaction records confidential within a set of organizations that belong to a channel, established as members. Each channel maintains a separate ledger and peers in the same channel share the same ledger

   (d) **Private Data Collections** [52]: Private data collections allow subsets of organizations within a channel to share confidential data. These collections enable the sharing of private data only with the organizations authorized to access it while storing a hash of the data on the channel's ledger. This mechanism ensures data confidentiality and integrity among specific members without exposing sensitive information to the entire channel.

2.2.1 Transaction Flow

Unlike most blockchain platforms, Hyperledger Fabric employs a transaction flow structure referred to as execute-order-validate, as shown in Figure 2.2. This structure consists of three distinct phases:

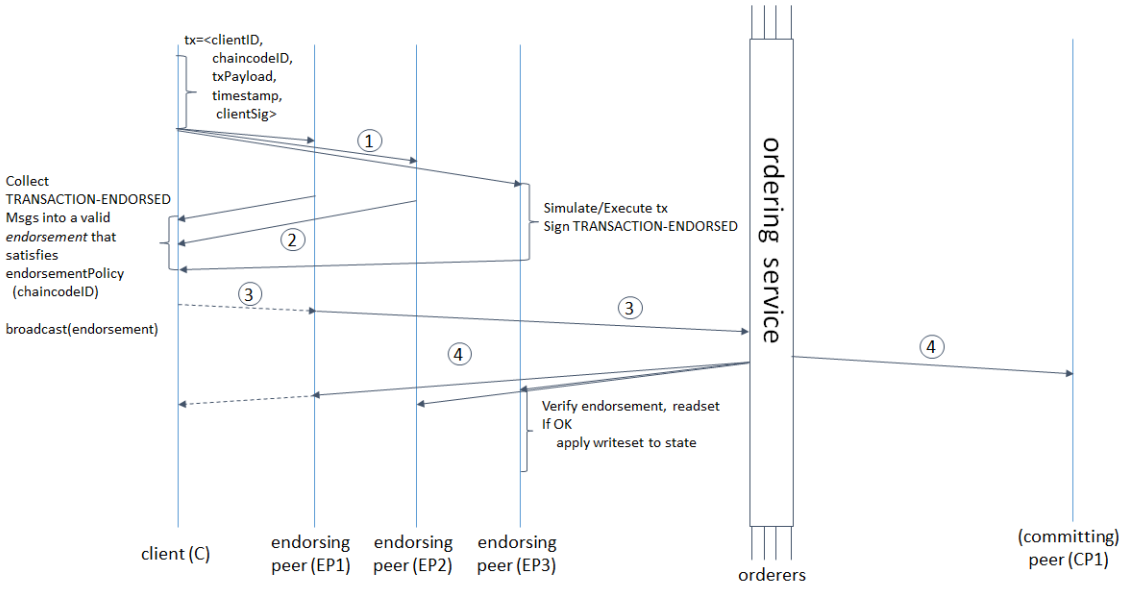

Figure 2.2: Transaction Flow (taken from [4])

1. **Execution Phase**: In the execution phase, clients propose transactions, which are then endorsed by a subset of peers. Each peer independently simulates the proposed transaction without updating the ledger. During this simulation, the peers execute the smart contracts (chaincode) to generate a read-write set, which contains the data read from and written to the ledger. The peers then sign this read-write set, producing an endorsement.

2. **Ordering Phase**: Once a transaction is endorsed, it moves to the ordering phase. In this phase, orderer nodes collect endorsed transactions and sort them into a definitive order. The transactions are grouped into blocks and distributed to all peers in the network. It is important to note that the orderers do not examine the transaction content for validity; they simply ensure the transactions are in a consistent order across the network.

3. **Validation Phase**: In the validation phase, each peer independently validates the transactions within the blocks received from the orderers. This phase includes two critical checks:

   (a) **Endorsement Policy Check**: Verifies that the transaction has the required endorsements as specified by the endorsement policy.

   (b) **MVCC (Multi-Version Concurrency Control) Check**: Ensures that the read set of the transaction has not changed since the transaction was endorsed. This prevents conflicts that could arise from concurrent transactions modifying the same data.

Only transactions that pass these validation checks are committed to the ledger, updating the world state. While the execute-order-validate architecture provides several benefits, such as enhanced security and flexibility in policy enforcement, it does not inherently offer better performance improvements compared to the traditional execute-order architecture. The validation phase can introduce additional latency due to the MVCC check, which ensures consistency but can slow

down the processing of transactions. A comparative study of various existing solutions and the rationale for selecting Hyperledger Fabric is detailed in Chapter 3.

2.3 Cloud Service Models

Cloud computing has revolutionized how businesses leverage technology by facilitating instant scalability of infrastructure and eliminating the need for traditional IT staff to manage physical hardware maintenance. Among the most common cloud service models—Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS)—businesses can select the model that best aligns with their operational requirements and technological objectives [16]. Below are descriptions of each:

1. **IaaS (Infrastructure as a Service)**: Enables clients to use virtualized resources provided by a cloud provider instead of maintaining physical servers on-premises. Clients manage virtual machines, networks, and software configurations, while the provider handles physical hardware maintenance. This model offers flexibility akin to traditional infrastructure ownership, reducing upfront investments and operational complexities.

2. **PaaS (Platform as a Service)**: Simplifies application development and deployment by abstracting away infrastructure management. Clients can develop, run, and manage applications without handling underlying hardware, operating systems, or network infrastructure. The provider manages resource provisioning, scaling, and maintenance, accelerating the software development lifecycle and freeing developers to focus on coding.

3. **SaaS (Software as a Service)**: Delivers applications over the Internet without local installation or maintenance. Unlike IaaS and PaaS, SaaS abstracts hardware, operating systems, and software maintenance from end-users, offering seamless access from any device with internet connectivity, reducing costs for updates, backups, and IT support. SaaS operates on subscription-based pricing, ensuring scalability and flexibility.

Sensefinity, a startup, has adopted cloud computing, specifically utilizing Google Cloud for its services as it does not require upfront investment. This project is entirely implemented on the Google Cloud platform, specifically within a Kubernetes cluster, which falls under the category of IaaS (Infrastructure as a Service). Kubernetes will be further explained in the following section.

2.4 Kubernetes

Kubernetes, commonly known as K8S [38], is an open-source container orchestration system that automates the deployment, scaling, and management of containerized applications. In this project, Kubernetes plays a pivotal role in deploying blockchain components, enhancing reliability and availability through containerization. By dynamically scaling resources based on demand,

Kubernetes ensures robust operation under varying conditions. It streamlines management processes, facilitating seamless deployment and operation critical for maintaining a high-performance blockchain network.

The core deployment units in Kubernetes, as outlined in Chapter 5.4, are:

1. **Pod**: A Pod is the smallest deployable unit in Kubernetes. It represents a single instance of a running process in a cluster and can encapsulate one or more containers. Containers within a Pod share the same network namespace, which means they can communicate with each other using localhost.

2. **Deployment**: A Deployment in Kubernetes is a higher-level abstraction that manages the deployment and scaling of multiple Pods. It enables the description of the desired state for the application, such as the number of replicas (Pod instances) and the container image version. Deployments provide features like rolling updates and rollback mechanisms to ensure that the application is running as expected.

3. **Service**: A Service is an abstraction that defines a logical set of Pods and a policy by which to access them. It provides a stable endpoint (IP address and port) that can be used to communicate with the Pods. Services enable load balancing across multiple Pods, and they abstract away the details of Pod IP changes, making it easier for other components to find and connect to the application.

4. **Persistent Volume (PV)**: A Persistent Volume is a piece of storage in the cluster that has been provisioned by an administrator. It is a resource in the cluster, much like a node is a cluster resource. PVs are used to provide durable storage for stateful applications. They exist independently of Pods and can be dynamically or statically provisioned.

5. **Persistent Volume Claim (PVC)**: A Persistent Volume Claim is a request for storage by a user. It is used by a Pod to request access to a specific amount of storage defined by a Persistent Volume. PVCs provide a way for developers to consume storage without worrying about the underlying details of how that storage is provisioned and managed.

Together, these components enable a robust and scalable platform for deploying and managing containerized applications in Kubernetes, which is the standard deployment approach for Sensefinity and its suite of services.

2.5 Hyperledger Caliper

Hyperledger Caliper [22] is an open-source benchmarking tool designed to assess the performance of various Distributed Ledger Technology (DLT) platforms, such as Hyperledger Fabric, Hyperledger Besu, and Ethereum. It simulates a network environment where multiple client nodes, referred to as workers, interact with the blockchain to execute transactions, including querying blockchain states or initiating actions that create or modify data.

The benchmark tests by Caliper are essential for ensuring that the blockchain system effectively handles the traffic load and processes all transactions required for the traceability system. Adjustable parameters such as transactions per second (TPS), total transactions, and test duration allow for comprehensive performance evaluations under diverse scenarios. Key performance metrics such as transaction throughput, latency, and resource utilization are systematically collected and analyzed by Caliper, providing valuable insights into the efficiency and scalability of DLT platforms.

This tool was selected due to its direct compatibility with Hyperledger Fabric, offering a straightforward approach to accurately benchmark the network's performance.

2.6 Summary

This chapter outlines the foundational technologies for a traceability system aimed at the Fundão cherry supply chain. It delves into blockchain technology, initially conceptualized by Satoshi Nakamoto in 2008 for Bitcoin, which ensures secure and immutable transactions through its decentralized, append-only transaction log. The discussion includes both permissionless blockchains, such as Bitcoin and Ethereum, which are open to public participation and use mechanisms like Proof of Work, and permissioned blockchains like Hyperledger Fabric, which offer greater control and efficiency through consensus algorithms suited for closed networks. Hyperledger Fabric's architecture, featuring nodes like peers, orderers, and clients, and mechanisms such as channels and private data collections, supports data confidentiality and transaction validation.

Additionally, the chapter covers the use of cloud service models—Infrastructure as a Service (IaaS), Platform as a Service (PaaS), and Software as a Service (SaaS)—with the project leveraging Google Cloud and Kubernetes (IaaS) to deploy and manage containerized blockchain components, enhancing scalability and reliability. Hyperledger Caliper is utilized for benchmarking various Distributed Ledger Technologies (DLTs), assessing performance through metrics such as transaction throughput and latency. Collectively, these technologies form a robust framework for developing a scalable, secure, and efficient traceability system for the Fundão cherry supply chain.

Chapter 3

Related Work

The related work section provides a focused analysis of distinct aspects of the research and decision-making process. The objective is to comprehend successful methods from other studies and tailor them to the specific domain. The sections are divided as follows:

- **Section 3.1**: Identifies the most suitable permissioned blockchain framework for the specific use case. The rationale for selecting Hyperledger Fabric will be subsequently elucidated.
- **Section 3.2**: Explores the consensus algorithms that can be utilized by Hyperledger Fabric, focusing on their performance and resource utilization.
- **Section 3.3**: Investigates the state databases that can be used with Hyperledger Fabric, based on benchmarks focused on write transaction throughput.
- **Section 3.4**: Examines research papers that focus on architectural aspects and ideas applicable to blockchain technology for product traceability.
- **Section 7.4**: Provides a summary of this chapter.

3.1 Permissioned Blockchain Frameworks

Polge et al. [44] conducted an in-depth comparative study focusing on five leading permissioned blockchain frameworks extensively utilized across industries: Hyperledger Fabric, Ethereum, Quorum, Multichain, and R3 Corda. Their analysis covered multiple critical dimensions including community engagement, performance metrics, scalability, privacy features, and adoption trends within industrial settings. Each framework was evaluated based on these criteria, providing a nuanced understanding of their respective strengths and limitations.

Hyperledger Fabric, as highlighted by Polge et al., stands out due to its decentralized architecture and versatility in supporting Dapps through smart contracts, or chaincodes. Fabric's design, hosted by the Linux Foundation, enables permissioned networks where nodes are categorized into clients, peers, and ordering service nodes, each fulfilling distinct roles in transaction processing and ledger maintenance. Notably, Fabric supports various consensus mechanisms, including

Practical Byzantine Fault Tolerance (PBFT) and Raft, offering flexibility in network governance without a native cryptocurrency, distinguishing it from public blockchain models.

In contrast, Ethereum, initially designed as a public blockchain platform, offers configurable permissioned setups suitable for enterprise applications. Polge et al. discuss Ethereum's transition from Proof-of-Work (PoW) to Proof-of-Authority (PoA) consensus mechanisms like Clique, reducing computational intensity while maintaining transaction integrity. Ethereum's Ethereum Virtual Machine (EVM) allows decentralized applications to execute across the network, supported by its native cryptocurrency Ether, which adds a layer of financial incentive and security within the ecosystem.

Quorum, developed by J.P. Morgan, emerges as another notable framework tailored for financial industries but adaptable to broader enterprise applications. Polge et al. underscores Quorum's enhancements over Ethereum, including privacy features that enable confidential transactions visible only to specified participants, and alternative consensus protocols such as Raft-based and Istanbul BFT. These enhancements bolster performance and security, which are crucial for financial applications requiring transaction confidentiality and regulatory compliance.

Multichain, a derivative of Bitcoin, emphasizes configurability and simplicity, allowing organizations to tailor blockchain parameters like access permissions, data privacy, and block sizes to suit specific business needs. Polge et al. note the evolution from Multichain 1.0 to 2.0, introducing Smart Filters to enable custom transaction validation rules, and expanding its utility beyond simple transaction tracking to more complex business logic implementation.

R3 Corda, developed by the R3 Consortium, focuses on permissioned environments requiring stringent identity verification through its "Know Your Customer" (KYC) principle. Polge et al. highlights Corda's architecture, which supports distributed applications (CorDapps) in Kotlin, emphasizing transaction validity and uniqueness verified by designated Notary nodes. Corda's design addresses specific use cases in finance and beyond, where transaction privacy and regulatory compliance are paramount.

The study's methodology involved evaluating each framework's community engagement through metrics like Github activity and social media presence, adoption rates based on industry reports like Forbes Blockchain 50, and performance metrics derived from a comprehensive academic literature review. Polge et al. assigned values from 0 to 5 to each metric in the comparative analysis, as summarized in Table 3.1. Challenges acknowledged include the reliance on outdated research, which complicates direct performance comparisons across frameworks due to evolving hardware capabilities and platform updates.

Hyperledger Fabric has been selected as the blockchain framework for this project, recognized as the second most prominent in the realm of permissioned blockchains according to Polge et al. Its widespread adoption ensures ample background knowledge and resources are accessible for system development consultations.

Fabric demonstrates robust performance metrics crucial for supply chain operations, such as low latency, high throughput, and scalability, which align with the interoperability team's require-

Criteria	Hyperledger Fabric	Ethereum	Quorum	MultiChain	R3 Corda
Adoption	3	5	1	0	2
Privacy	5	0	4	4	3
Scalability	4	5	0	0	1
Throughput	5	4	4	4	1
Latency	5	2	2	3	4

Table 3.1: Comparison of Permissioned Blockchain Frameworks (adapted from [44])

ments outlined in Chapter 4.

Moreover, Fabric's strong emphasis on privacy and security complements its modular architecture. This architecture supports containerized applications like Kubernetes, enabling blockchain components to function as independent modules or microservices. This modularity enhances flexibility and scalability in deploying and managing blockchain solutions.

Additionally, Sensefinity's backend projects leverage Golang for its cloud-native capabilities, further reinforcing Hyperledger Fabric's suitability due to their compatibility.

3.2 Consensus Algorithms

Yang et al. [54] present a comprehensive analysis of consensus algorithms for running Blockchain as a Service (BaaS) in cloud environments using the Hyperledger Fabric framework. Their study focuses on measuring performance and resource consumption. The consensus algorithms chosen for the analysis include Kafka and Raft, which are officially supported by Fabric, and PBFT (Practical Byzantine Fault Tolerance), which is not officially supported but serves as the foundational algorithm for other variants. These algorithms are detailed below:

3.2.1 Kafka

Structure and Operation

Kafka, developed by LinkedIn, consists of producers, consumers, and broker nodes. Producers create data, and consumers read it via broker nodes that store the data. In Hyperledger Fabric, broker nodes manage transaction replication. An orderer node receives client transactions and sends them to a leader broker node, which orders and replicates the transactions. Orderer nodes periodically pull these transactions, and blocks are created once a threshold is met.

Fault Tolerance

Kafka's fault tolerance depends on the number of broker nodes (n) and a parameter (m). It can tolerate faults up to $n - m$. Apache Zookeeper monitors and selects new leaders for orderer and broker nodes when failures occur.

3.2.2 Raft

Structure and Operation

Raft employs orderer nodes with a leader and followers. The leader orderer node receives client transactions, accumulates them until a batch size is reached, and then forms a block. This block is replicated to follower nodes, which store it and confirm receipt. Once all followers acknowledge the block, it is distributed to peer nodes.

Fault Tolerance

Raft's fault tolerance is based on the number of orderer nodes (n) and $m = \left\lceil \frac{n+1}{2} \right\rceil$. It can tolerate faults up to $\left\lfloor \frac{n-1}{2} \right\rfloor$, typically configuring an odd number of nodes for optimal tolerance. For example, with five orderer nodes, Raft can handle up to two node failures.

3.2.3 PBFT (Practical Byzantine Fault Tolerance)

Structure and Operation

PBFT addresses Byzantine faults through a three-phase process: pre-prepare, prepare, and commit. In the pre-prepare phase, the primary node (leader) sends a pre-prepare message to all replica nodes. During the prepare phase, each replica node broadcasts a prepare message to all other nodes upon receiving the pre-prepare message. In the commit phase, nodes send commit messages after receiving prepare messages from a majority. A transaction is committed when a node receives a majority of commit messages.

Fault Tolerance

PBFT can tolerate up to f faulty nodes in a system with $3f + 1$ nodes, ensuring system correctness even if some nodes fail or act maliciously. For instance, in a seven-node system, PBFT can tolerate up to two faulty nodes.

3.2.4 Evaluation Results

The results include metrics such as throughput (transactions per second), latency (time to commit a transaction), CPU usage, and bandwidth usage. These metrics were evaluated across various configurations: variations in batch sizes, changes in node numbers, adjustments in payload sizes, and simulations of fault scenarios in both broker nodes (Kafka) and orderer nodes (Raft and PBFT). The summarized results are as follows:

Batch Size Experiments

Kafka and Raft demonstrated a consistent throughput of around 829 TPS across batch sizes ranging from 10 to 200. Both algorithms saw slight increases in throughput with smaller batch sizes due to reduced processing overheads. However, they experienced declines when batch sizes exceeded 100, leading to increased latency. In contrast, PBFT maintained lower overall throughput,

ranging from 625 to 678 TPS with minimal fluctuations, reflecting its more intensive verification process involving multiple orderer nodes.

Node Number Changes

Kafka and Raft maintained stable throughput performance of approximately 829 TPS across node configurations ranging from 3 to 10 orderer nodes. Neither algorithm exhibited significant changes in throughput as the number of nodes increased, showing consistency throughout. In contrast, PBFT showed a marked decline in throughput with increasing orderer nodes. Specifically, PBFT's throughput decreased by 70% when scaling from 4 nodes to 10 nodes, highlighting its sensitivity to larger node counts compared to Kafka and Raft.

Payload Size Adjustments

Adjustments in payload size (1 KB, 2 KB, 4 KB, 8 KB) resulted in varying impacts on CPU usage and latency for the consensus algorithms. Kafka and Raft demonstrated moderate increases in CPU usage and latency with larger payloads. For instance, Kafka's CPU cycles rose by 11.3% at a 4 KB payload size, while Raft's CPU usage increased by 35% under the same conditions. PBFT experienced a substantial 77.5% decrease in throughput and a 3.8x increase in latency at a 4 KB payload size. Bandwidth usage increased steadily for Kafka and Raft but remained relatively stable for PBFT.

Fault Scenarios

In fault scenarios, Kafka's throughput dropped by 87% when 8 broker nodes failed, surpassing its fault tolerance limit of 7 nodes. Raft, within its fault tolerance limit of 4 orderer nodes, maintained stability; however, exceeding this limit caused a significant 76% drop in throughput. PBFT showed slight improvements within its fault tolerance limit of 3 orderer nodes but suffered a significant 95.8% decrease in throughput and increased latency beyond this limit.

3.2.5 Conclusion

The selected algorithm for this project is Raft, which is also the recommended algorithm for version 2.5 of Hyperledger Fabric, the version the project utilizes. Evaluation results show that both Kafka and Raft demonstrate strong performance across various test categories, with Raft standing out in fault tolerance scenarios.

The choice of using Raft is supported by the fact that Kafka and ZooKeeper are not designed for large networks and are best run within a single organization's tight host group [29]. In contrast, Raft allows each organization to have its ordering nodes, leading to a more decentralized system. With Kafka, all nodes connect to a single Kafka cluster controlled by one organization. Additionally, Raft is easier to set up because it does not require a ZooKeeper ensemble.

3.3 State Databases

Laishevskiy et al. [39] explore the optimization of the State Database (StateDB) within Hyperledger Fabric (HLF). StateDB plays a critical role in HLF by storing the current state of the blockchain network and is essential for transaction validation and peer interaction. The study compares the current StateDB implementations—GoLevelDB and CouchDB—and proposes evaluating alternative databases such as RocksDB, Boltdb, and BadgerDB to enhance performance.

CouchDB is a NoSQL database that uses JSON for documents, JavaScript for MapReduce indexes, and regular HTTP for its API. It provides a rich query language and supports complex queries, indexing, and data versioning. These features make CouchDB particularly useful for projects with high transaction volumes that require these capabilities. Despite a slight trade-off in performance compared to LevelDB, opting for CouchDB aligns with the project's requirements, prioritizing its extended capabilities for more effective data handling.

On the other hand, LevelDB is an embedded key-value store database that provides ordered mapping from string keys to string values. It uses Log-Structured Merge (LSM) trees for efficient writes and reads but does not support rich queries like CouchDB. LevelDB's ease of use and speed make it a preferable choice for projects with a low volume of transactions.

These databases are the currently supported options in Hyperledger Fabric, each serving different needs. However, to determine if these are the best options, this study evaluates and compares embedded key-value store databases like RocksDB, BoltDB, and BadgerDB. The study benchmarks focused on write transaction throughput across different implementations, covering various transaction sizes (from 1KB to 64KB), as shown in Figure 3.1.

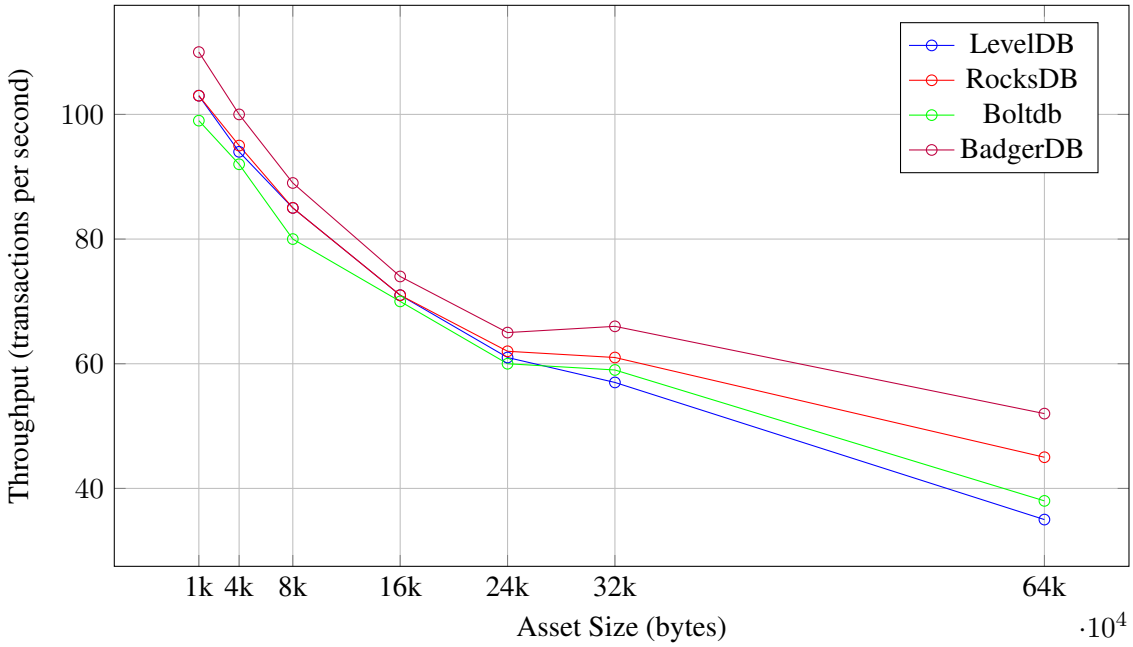

Figure 3.1: Throughput Comparison of StateDBs in Hyperledger Fabric (adapted from [39])

BadgerDB emerged as the preferred choice among the potential StateDBs. It demonstrated a

significant advantage in the StateDB write performance benchmarks, especially for write values of 64KB, where its transactions per second (TPS) were almost 1.5 times higher than those of LevelDB. Additionally, BadgerDB offers features not available in LevelDB, such as guaranteeing ACID properties and enabling custom queries in HLF through an additional tool.

3.4 Traceability Systems

Dasaklis et al. [14] literature review on implementing blockchain for supply chain traceability guided the selection of research papers that contributed to the development of the system architecture solution. The choice of these articles was largely based on studies that address significant challenges and offer practical insights into the real-world application of traceability systems.

Premarathna's [45] research on food waste from farm to consumer in Sri Lanka provides a thorough overview of the fabric network architecture, detailing the purpose of creating individual channels for each party. This design ensures data privacy for each participant involved in the network. Premarathna's research guided the setup of the blockchain in a way that ensures each participant's information remains accurate and private.

Uddin et al.'s [51] study on Medledger, focusing on drug traceability in the pharmaceutical industry, highlights the crucial role of digital certificates in authenticating stakeholders and ensuring robust security within the blockchain framework. Their research underscores the necessity for a dedicated certificate server to effectively manage identities on the blockchain, thereby guaranteeing secure and trustworthy transactions across the drug supply chain. This study influenced the system architecture by advocating for the implementation of certificate servers for each organization within the consortium, enhancing identity management efficiency.

Lin et al. [40] have made significant advancements in smart agriculture by integrating blockchain technology with IoT devices for enhanced food-tracking capabilities. Their development minimizes human involvement, thereby enhancing the accuracy and reliability of blockchain-stored data. This shift towards sensor-driven data fosters trust and transparency among stakeholders, ensuring a more accountable food-tracking process across the supply chain. Their work inspired Sensefinity to monitor supply chain parameters through sensors, effectively reducing errors in data integration into the blockchain for this project.

Yang et al.'s [55] system for traceability in agricultural product farming suggests using a hybrid 'database + blockchain' approach to manage large volumes of data, such as sensor data. They found that this approach reduces strain on the blockchain system, improving the efficiency of information queries. Their emphasis lies in leveraging an external database for efficient storage of comprehensive data, selectively providing essential information while preventing unnecessary clutter on the blockchain. In this hybrid model, cryptographic pointers on the blockchain reference the actual data stored in the database. This integration approach is pivotal for incorporating Sensefinity's sensor data, which tracks the entire journey of cherry packages from farm to end consumer, into the blockchain during the project's upcoming phase.

Tzenetopoulos et al. [50] study introduces a novel blockchain-based framework, 'HLF-Kubed,'

for monitoring edge devices. It utilizes the Kubernetes container orchestrator alongside the Hyperledger Fabric blockchain framework. This approach underscores the integration of distributed ledger technology with existing container management systems to boost the scalability and reliability of blockchain systems. The study influenced the containerized architecture of the system and emphasized the importance of chaincode as an external service in this type of architecture.

In summary, these studies have been instrumental in shaping the system. For a concise overview, refer to Table 3.2, which outlines the contributions to the traceability systems described above.

Author	**Contribution**
Premarathna's [45]	Detailed fabric network architecture for food traceability, ensuring data privacy and accuracy.
Uddin et al. [51]	Emphasized digital certificates for secure identity management in pharmaceutical traceability.
Lin et al. [40]	Developed blockchain and IoT solution for precise food tracking, reducing human involvement.
Yang et al. [55]	Proposed a hybrid database-blockchain approach for efficient data handling in agricultural traceability.
Tzenetopoulos et al. [50]	Introduces blockchain framework for edge device monitoring, integrating Kubernetes and Hyperledger Fabric, enhancing scalability.

Table 3.2: Traceability Systems and Their Contributions

3.5 Summary

The related work section explores critical dimensions of blockchain technology pertinent to the project, including permissioned frameworks, consensus algorithms, state databases, and traceability systems. Polge et al. offer a comparative analysis of major permissioned blockchains, identifying Hyperledger Fabric as the most suitable framework for its modular architecture and robust performance. Yang et al. provide an in-depth evaluation of consensus algorithms, highlighting Raft for its superior fault tolerance and ease of use over Kafka and PBFT. In state databases, Laishevskiy et al. find BadgerDB to be the optimal choice for its high performance and advanced features compared to GoLevelDB and CouchDB. Lastly, Dasaklis et al.'s review on traceability systems, alongside contributions from Premarathna, Uddin, Lin, Yang, and Tzenetopoulos, informs the system's design, emphasizing privacy, secure identity management, and efficient data handling in supply chain traceability.

Chapter 4

Use Cases and Requirements

To identify the functional and non-functional requirements of this project, it is essential to understand the Cerejas do Fundão supply chain in its entirety, including all processes. By thoroughly examining these processes, the requirements can be accurately identified. This chapter is divided into the following sections:

- **Section 4.1**: An overview of the organizations within the Cerejas do Fundão supply chain and their respective roles.
- **Section 4.2**: Analyzes the processes carried out by each organization within the Cerejas do Fundão supply chain.
- **Section 4.3**: Describes the functional and non-functional requirements of the project.
- **Section 7.4**: Provides a summary of this chapter.

4.1 Supply Chain Stakeholders and Roles

The Cereja do Fundão supply chain, which revolves around the cherries from the Fundão region, involves several key individuals and organizations as illustrated in Figure 4.1.

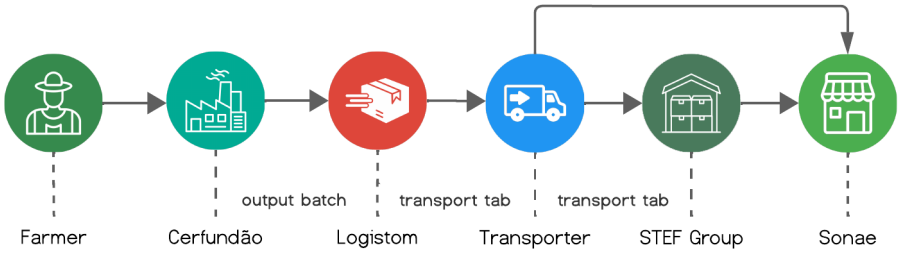

Figure 4.1: Cereja do Fundão Stakeholders

1. **Farmers**: These individuals are responsible for the entire cherry production process, starting from cultivation in the fields. After growing the cherries, they transport them to Cerfundão. This group forms a vital part of the cherry growers' community in the Fundão region.

2. **Cerfundão**: Serving as the quality control hub, Cerfundão ensures that each batch of cherries undergoes a rigorous industrialization process to guarantee their quality. These cherries are ultimately certified, with a quality assurance certificate included on the packaging, affirming their high standard.

3. **Logistom**: This entity manages all aspects of transportation logistics, often subcontracting other transport services to facilitate the delivery of cherries.

4. **Transporters**: As a subcontractor for Logistom, the transporter is equipped with refrigerated truck containers essential for transporting the cherries at cold temperatures to maintain their quality. Depending on the route's distance and the truck drivers' hours, the cherries might be delivered directly to the retailer or routed through distribution warehouses.

5. **STEF Group**: As a leader in the European food supply chain, the STEF group possesses several distribution warehouses across the country. They also provide specialized refrigerated trucks with containers, crucial for the cold chain management of perishable goods like cherries. In some cases, STEF may also serve as a subcontractor for Logistom in the transportation process.

6. **Sonae**: A multinational business group and retailer, Sonae purchases cherries from Cerfundão and distributes them through its network of supermarkets, making the cherries available to consumers.

Not all of these organizations will be incorporated into the blockchain system, including farmers, the STEF group, and the transporters. Farmers tend to stick to traditional methods and are less inclined to adopt technology, especially in Portugal where they are among the oldest in Europe and may have limited education [32]. To address this, Cerfundão has formed a group of farmers in the region, and the tracking of cherries will begin solely at the quality control center. The STEF group's role in the supply chain is indirect. They only provide temporary storage for cherries before they are transported to other destinations. Because they do not play a direct role in the cherry's journey, they are excluded from the blockchain system. The transporters are not included because there are various ones (Logistom works with around 15 types of different transporters), making it difficult to implement the system for each one. Ideally, each organization should contribute a peer, and perhaps in the future, those that are excluded now can participate. However, for these reasons, they are currently excluded.

4.2 Supply Chain Process Analysis

Figure 4.2 represents the processes in a BPMN style [2], illustrating the Cereja do Fundão supply chain. It details the processes each entity performs as a package of cherries is handled until it reaches the retailer that bought the cherries. The scenario assumes that the cherries are bought by Sonae from Cerfundão. What happens after the retailer receives the cherries (such as which

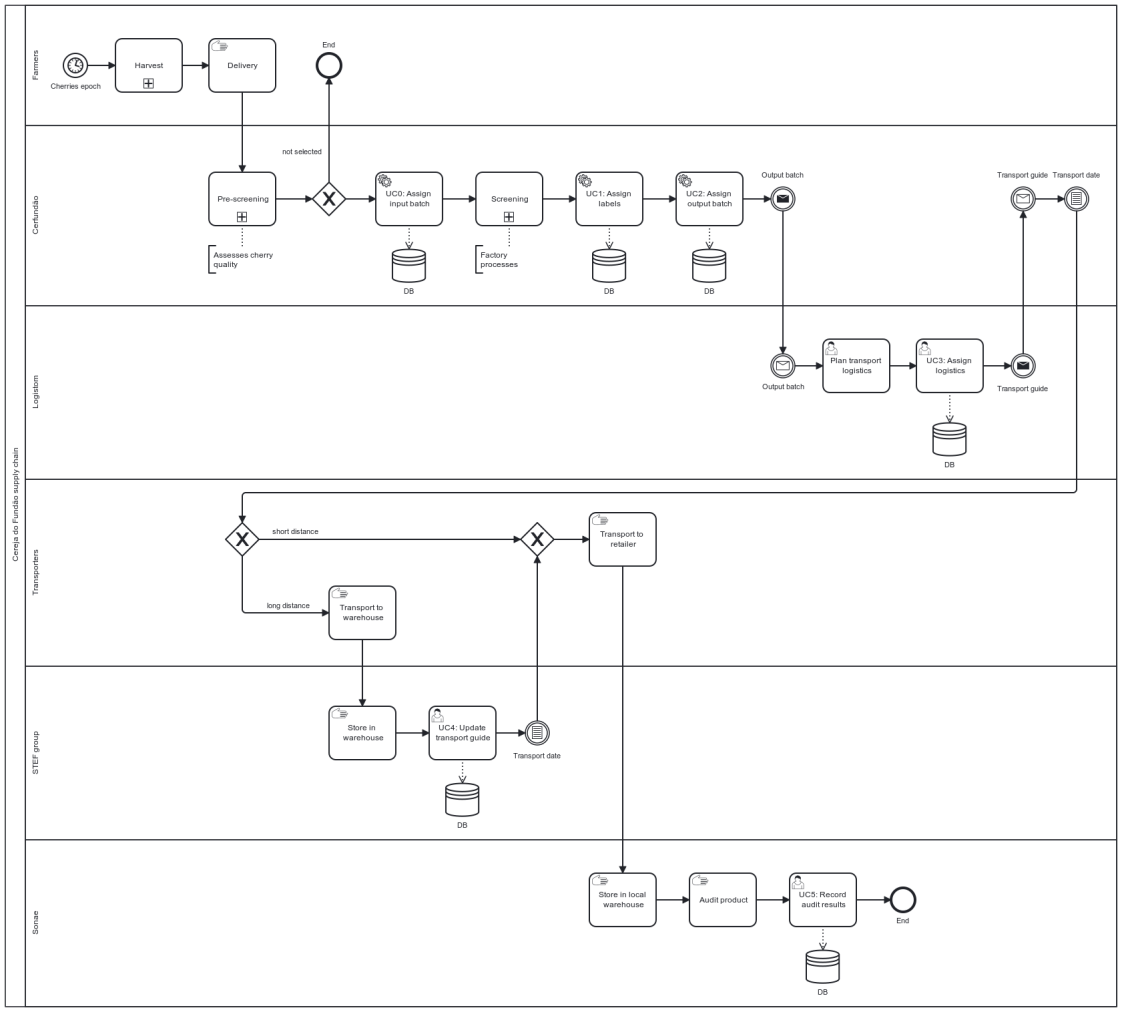

Figure 4.2: BPMN Diagram of the Cereja do Fundão Supply Chain

supermarket they go to or how they are handled) is not defined here, as it is not a requirement of this project for now.

The cherries' journey begins with the farmers during the harvest season. Once the cherries are ready, the farmers harvest and deliver them directly to Cerfundão. This direct delivery is possible because both the farmers and Cerfundão are located in the Fundão region, and the farmers are part of a group formed by Cerfundão. Upon arrival at Cerfundão, the cherries undergo a pre-screening process. During this stage, the cherries are filtered to ensure only the good ones proceed. Any substandard cherries are returned to the farmers. The cherries that pass the pre-screening are recorded in Cerfundão's system, noting the quantity and type of cherries received. Next, the cherries undergo various processing and industrialization steps at the factory to ensure their quality. This includes cleaning, sorting, and other quality assurance processes. Once these processes are completed, the cherries are packaged and labeled into small packages. The details of the packaged cherries are recorded in the system, including information about the cherries in each package. These packages are marked with a symbol of guarantee and quality certification, making them ready to be sold in supermarkets. Assuming the cherries are already bought, an output batch

is created that contains multiple packages of cherries corresponding to the purchase by Sonae. This output batch is associated with a specific number of packages, clearly indicating which packages are included in the batch. This information is then recorded in the Cerfundão system. All data recording into the Cerfundão system is performed by specialized industrialization machines. These machines automatically capture and log the necessary information without requiring a user interface for manual data entry.

Transportation logistics are managed by Logistom, which utilizes information provided by Cerfundão regarding the output batch to facilitate the delivery of cherries. This information is essential for planning effective transportation logistics. Currently, the process relies on exchanging emails and contacting a network of transporters to establish the most efficient transport plan. Each transportation company maintains this information on its respective platforms, as Logistom lacks a centralized system for this purpose. For this pilot, STEF is utilized as both the transporter and warehouse, utilizing their system. After finalizing the transportation plan, a guide is sent back to Cerfundão, detailing the timing of the cherry shipments—typically scheduled for the following day to ensure the perishable goods spend minimal time in warehouses.

Transporters collect the cherries from Cerfundão on the agreed transport date. Depending on the distance and kilometers traveled, drivers may need to pause the shipment in warehouses before continuing or proceeding directly to the retailer that purchased from Sonae. If the cargo passes through STEF, it is stored there, with STEF only permitted to store the cargo without handling it. Subsequently, STEF updates the transport guide in their system to indicate the cargo's arrival at the warehouse using their proprietary system and the system previously utilized by Logistom to arrange transportation. The transportation process resumes when the shipment is retrieved from the STEF warehouse, either by the same transporter or another (depending on arrangements made by the logistic company, Logistom). The cherries are then delivered to Sonae's local warehouses, where the products are stored before being distributed to supermarkets.

Upon receiving the cherries, Sonae conducts an audit to ensure the products meet their quality standards. The results of this audit are recorded in their system, documenting which products are accepted into their inventory and which are rejected. This auditing process is crucial for maintaining quality control before the cherries are further processed or sent to supermarkets. Processes beyond this point are irrelevant to this project.

4.3 Emerging Requirements

In the BPMN diagram described above, four types of assets are identified:

1. **Information about cherries**: This is divided into input batches, labeling, and output batches inserted by Cerfundão.

2. **Transporter logistics information**: This includes details about the transportation of cherries inserted by Logistom.

3. **Payment information**: This includes details about payments made by Sonae to Cerfundão.

4. **Retailer information**: This contains audit details related to the retail process inserted by Sonae.

Retailer information is excluded for now as it requires further requirements gathering from Sonae. The primary goal is to ensure that only relevant information is stored on the blockchain, given its slow performance with large data volumes. For example, if a batch of cherries fails pre-screening before entering the supermarket, this information is not recorded on the blockchain. Only cherries suitable for sale are documented, preventing the blockchain from being overloaded with unnecessary data. However, this information is still recorded in the organization's system. Similarly, not all cherry-related information, such as input batches, labeling, and output batches, is recorded on the blockchain. Instead, a single cherry asset includes only the most critical information from these data types stored in Cerfundão's system. The same approach applies to transportation logistics information and payment contracts. The essential aspect is to include references, such as IDs, to the real data in the organization's system. This allows for identifying any discrepancies by comparing the blockchain records with the systems used by each organization. The data structures for these assets are specified in Appendix A.

Cerfundão will integrate its system information into the blockchain using APIs. Given Cerfundão's automated system, which allows machines to insert data automatically, this integration will occur after an output batch of cherries is ready to leave the factory and the barcode on the cherries has been registered. Logistom requires a new system for integration since it currently relies on emails and Excel documents to store data. Sonae employees interact with GUIs to input their data into their respective systems and already use APIs for this purpose. Since immediate system integration is not feasible, data will initially be collected manually through Google Forms filled out by employees of each organization for the identified use cases. This data will then be subsequently inserted into the blockchain. By adopting this form-based data collection method, the pilot project can commence without requiring immediate and extensive integration of each organization's systems. This data will then be visualized in the Web Asset Management platform from Sensefinity, where each consortium member will have an account to access the data, authenticated with certificates from the blockchain.

4.3.1 Functional Requirements

For this pilot, each organization will have two user roles: members and administrators. Members have permissions for read and write operations in the blockchain, whereas administrators possess administrative privileges in addition to the permissions of members, allowing them to perform all operations that members can. These roles will be used by organizational users from Cerfundão, Logistom, and Sonae, each authenticated with certificates recognized by the blockchain. The functional requirements for each organization result from the use cases illustrated in Figure 4.3 and are elaborated upon below.

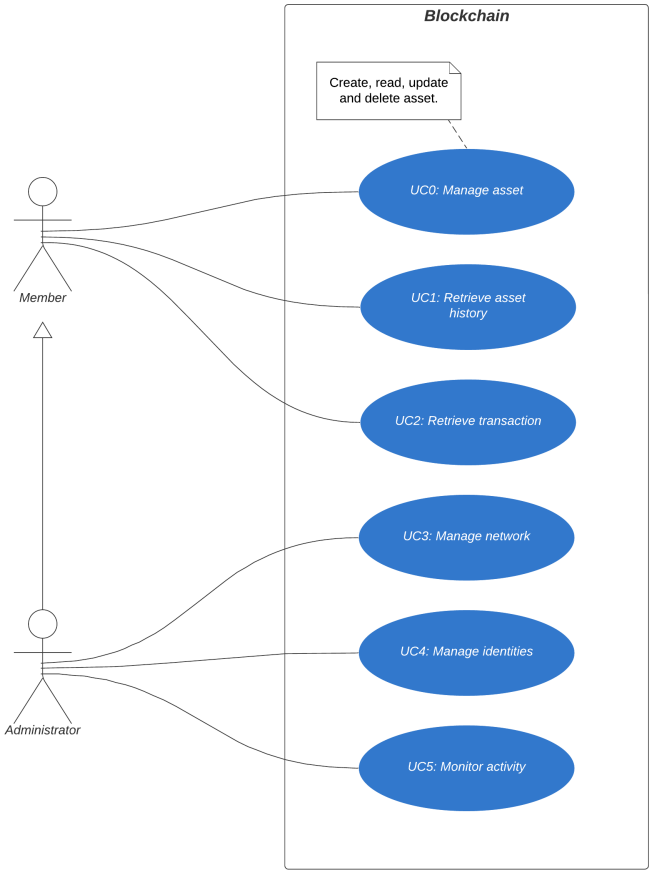

Figure 4.3: Use Case Diagram

1. **Manage Asset**

   - **Description:** Members can perform CRUD operations (Create, Read, Update, Delete) on assets using smart contracts and an API. The CRUD matrix in Table 4.1 illustrates the specific permissions granted to each organization regarding different asset types:

Assets	Cerfundão	Logistom	Sonae
Cherries	CRUD	R	
Logistics	R	CRUD	R
Payments	RU		CRUD

Table 4.1: CRUD Matrix for Supply Chain Assets

2. **Retrieve Asset History**

   - **Description:** Members retrieve the complete history of an asset, including all modifications made over time via an API.

3. **Retrieve Transaction**

- **Description:** Members access blockchain-specific details related to asset transactions, such as block identification, hashes, and responsible MSPs via an API.

4. **Manage Network**

   - **Description:** Administrators manage blockchain network tasks like channel provisioning, organization additions, and chaincode operations using a CLI.

5. **Manage Identities**

   - **Description:** Administrators handle blockchain identities through a CLI, including identity creation, updates, deletions, and certificate issuance.

6. **Monitor Activity**

   - **Description:** Administrators monitor blockchain network metrics through a dashboard, tracking transaction response times, node resource consumption, and overall network activity.

4.3.2 Non-functional Requirements

The non-functional requirements for this project adhere to the ISO/IEC 25010 standard [15], which defines quality characteristics and criteria for evaluating software products. The non-functional requirements of this project are:

1. **Performance**

   (a) **Scalability**: The system should handle increased loads as more participants and data join the blockchain network.

   (b) **Availability**: The blockchain system, deployed on Kubernetes in the cloud, must ensure high availability and reliability with minimal downtime.

   (c) **Efficiency**: The system must process transactions efficiently to maintain supply chain traceability, recording at least 60 transactions per minute with low-latency communication.

2. **Security**

   (a) **Data Integrity**: Blockchain technology must ensure data cannot be tampered with, preserving record integrity.

   (b) **Access Control**: Only authorized entities should access specific blockchain data, protecting sensitive information.

   (c) **Confidentiality**: The permissioned blockchain must safeguard private data, accessible only to authorized network participants.

(d) **Secure Communication**: All communication channels and protocols used by the system shall be secured using Transport Layer Security (TLS).

3. **Usability**

   (a) **Interoperability**: The system should integrate seamlessly with existing consortium systems through APIs.

   (b) **Simplicity**: The system should be easy to use, reducing the time and effort required to perform tasks.

4. **Reliability**

   (a) **Fault Tolerance**: The system must continue operations smoothly despite component failures.

   (b) **Consistency**: Data consistency across all blockchain nodes must be maintained.

   (c) **Resilience**: The system should remain operational if an organization exits, ensuring critical components are redistributed or replaced as needed.

5. **Maintainability**

   (a) **Modularity**: The design should be modular for easy updates, maintenance, and scalability.

   (b) **Documentation**: Provide comprehensive documentation for system components to aid future maintenance.

4.4 Summary

This chapter outlines the use cases and requirements for integrating a blockchain system into the Cerejas do Fundão supply chain. The supply chain involves key stakeholders: farmers who harvest and deliver cherries to Cerfundão, which processes and certifies the cherries, Logistom which manages transportation logistics, and Sonae, the retailer distributing the cherries through its supermarkets. Certain stakeholders like farmers, STEF, and various transporters are excluded from the blockchain due to practical constraints such as technology adoption and system integration.

It details the process flow from cherry cultivation to delivery, including harvesting, quality screening, processing at Cerfundão, and transportation managed by Logistom and STEF. The blockchain will capture essential data such as cherry batches, logistics, and payments, focusing on efficiency by excluding less critical information. Functional requirements include managing and retrieving asset data, tracking history, and overseeing network operations, while non-functional requirements ensure scalability, security, efficiency, and reliability. Next, a solution will be proposed to meet these needs.

Chapter 5

Architecture

In this chapter, an architectural solution that aims to fulfill the goals and requirements outlined in the previous chapter is presented. The chapter is systematically organized into several sections, each detailing a specific aspect of the proposed architecture:

- **Section 5.1**: Presents the ideal architecture to demonstrate a fully decentralized blockchain system, focusing on the components each organization in the consortium should possess.
- **Section 5.2**: Outlines all the components required by each organization for the first pilot phase, highlighting the differences between the ideal and more accessible architecture.
- **Section 5.3**: Details how the blockchain network should be arranged to ensure that information between different organizations and assets remains private and confidential.
- **Section 5.4**: Illustrates the organization of the previously described components within a Kubernetes environment to enhance the availability of the deployed system.
- **Section 5.5**: Provides a summary of this chapter.

5.1 Optimal Component Distribution

The diagram (5.1) shows the optimal distribution of components among organizations within a consortium to create a fully decentralized system. According to the Hyperledger Fabric deployment guide [30], an optimal organization independently manages both application and orderer roles, maintains high availability with multiple instances, uses separate CAs for identity and TLS, and manages resources effectively. This configuration prevents any single organization from holding all critical components, thereby enhancing system resilience—a key non-functional requirement—and supporting decentralized principles.

5.1.1 Distribution Strategy

Distributing roles across multiple organizations prevents the risk of disruption if one organization exits the consortium. For instance, if an organization that solely manages orderers exits, the application organizations relying on its orderers would cease to function. In Hyperledger Fabric,

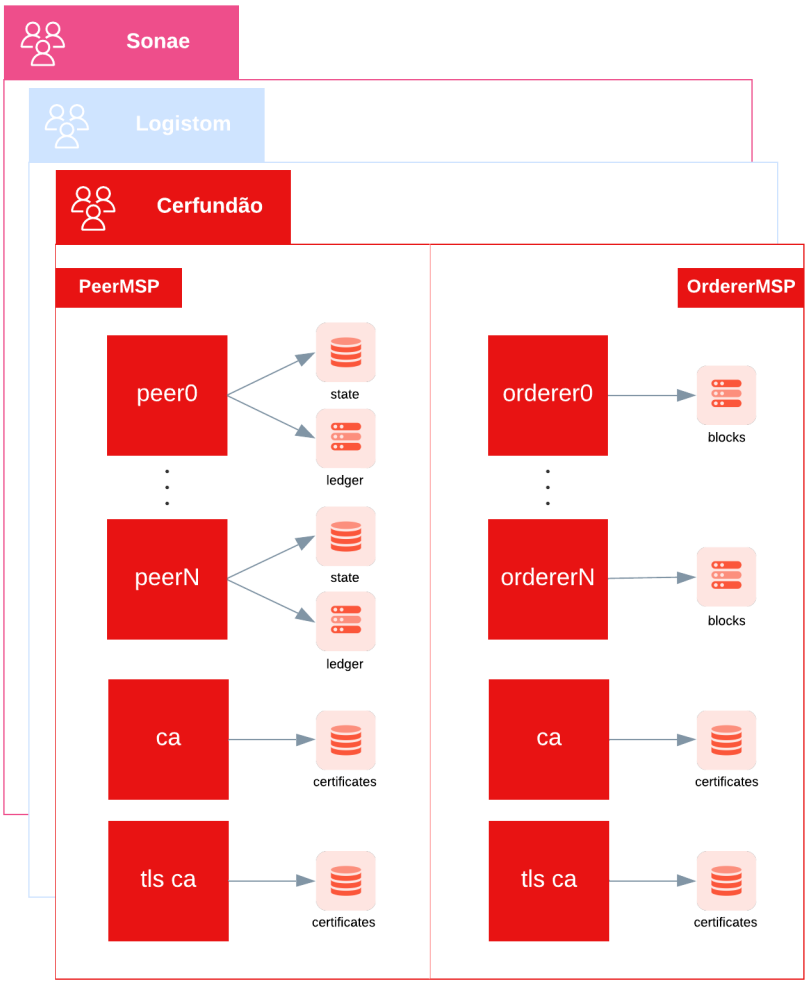

Figure 5.1: Optimal Component Organization

although technically possible, it is discouraged for an organization to handle both ordering and application roles due to safety issues like double spending attacks [8]. Separating ordering and application functions ensures that ordering nodes can't create valid transactions, and application users can't manipulate blocks. To address this, each organization operates as two separate entities, CerfundaoPeerOrg and CerfundaoOrdererOrg, each with a separate Membership Service Provider (MSP) like PeerMSP and OrdererMSP. Each organization must connect to an MSP, overseeing identity management within the network. Separate MSPs ensure peers from different organizations do not recognize each other as part of the same entity.

5.1.2 Certificate Authorities (CAs)

Each Managed Service Provider (MSP) needs its own dedicated Certificate Authority (CA) servers for generating certificates. Two types of CAs should be deployed: an organizational CA for creating organizational and node identities, crucial for access control in the blockchain (a non-functional requirement), and a TLS CA for issuing TLS certificates, ensuring secure communi-

cation between nodes in production (a non-functional requirement). Having separate CAs for organizational and TLS purposes establishes independent chains of trust, minimizing the impact of potential security breaches. Each CA stores its data in separate databases to ensure efficient management and reduce vulnerabilities. Servers managing organizational and TLS certificates are usually single-instance due to their minimal usage and non-critical nature, though exceptions can apply.

5.1.3 Peers and Orderers

Each participating organization contributes at least one peer to the network. Additional peers can be either replicated or independent. Replicated peers may maintain snapshots of ledger data from different channels, thereby enhancing fault tolerance, a critical non-functional requirement, by reducing the risk of compromising all data through a single peer. Each peer should have its distinct database to store the world state and a file storage system for ledger storage, allowing efficient access to the current value of an object without traversing the entire blockchain. Each organization also contributes at least one orderer, which orchestrates transactions independently. Orderers need a file storage system to maintain transaction history, ensuring that all peers have consistent records of how consensus is achieved among network participants.

5.2 Blockchain Structure

The architecture discussed in the previous chapter represents the ideal implementation but involves significant costs that may limit accessibility for many organizations. To address these financial constraints, a simplified variation has been implemented for this phase of the project.

Figure 5.2 illustrates this cost-effective architecture, which modifies the original design to reduce development and configuration expenses. Despite these simplifications, this architecture still forms the foundation for meeting all functional requirements by establishing essential blockchain infrastructure to effectively manage transactions, identities, network operations, and monitoring activities.

5.2.1 Peers and Orderers

The decision to deploy only one peer and one orderer per organization was driven by cost considerations, as peers consume significant resources in managing chaincodes and ledgers. Following this project phase, decisions regarding potential adjustments, such as adding more peers for load balancing or implementing snapshots, will be guided by the outcomes observed in the pilot environment.

Sensefinity, despite not being directly part of the consortium, sought involvement in the blockchain network. It is the only organization holding more than one orderer, a decision made based on the recommendation to maintain at least five nodes in a production network for enhanced fault tolerance (a non-functional requirement), which represents the current number in use. This ensures that

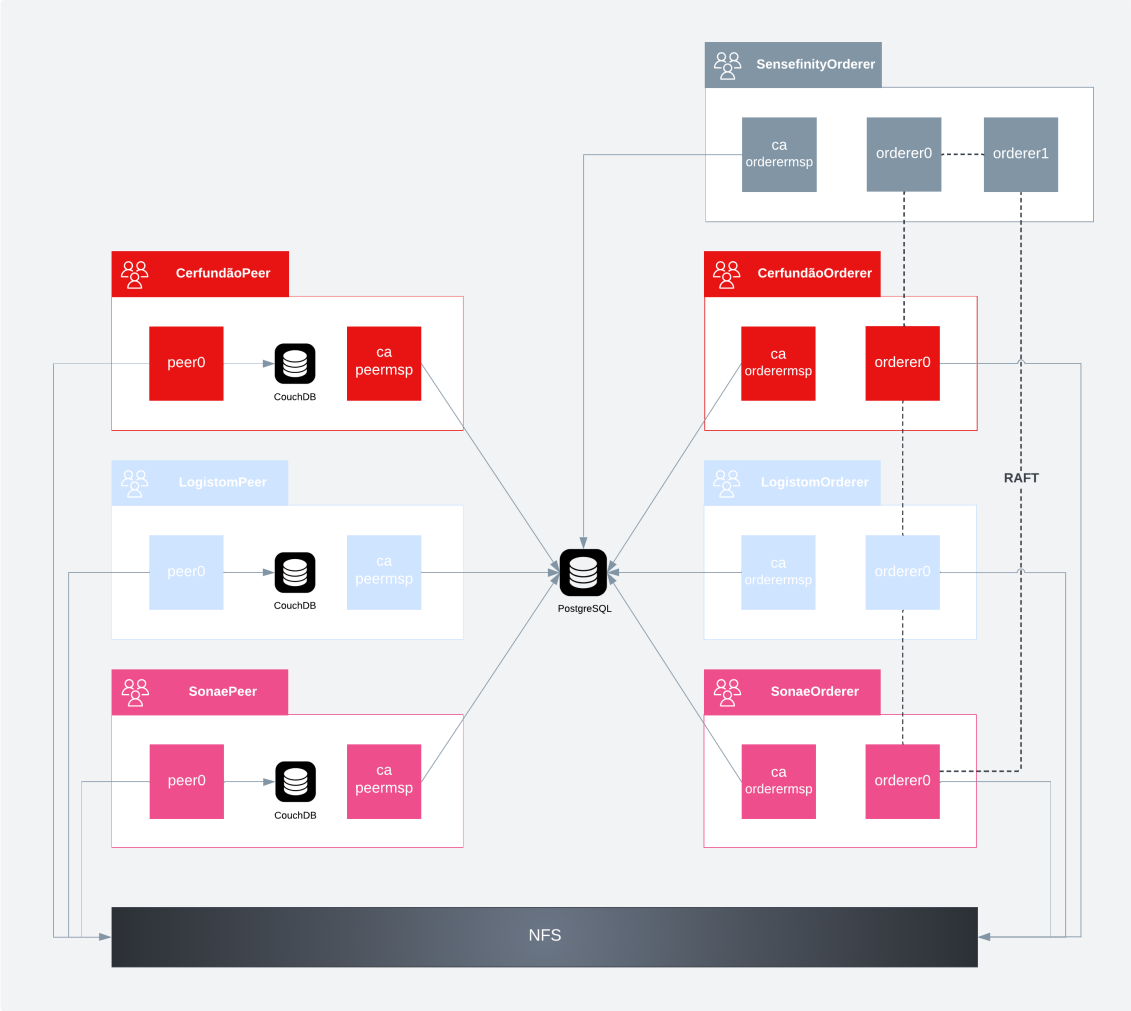

Figure 5.2: Blockchain Structure

the system can sustain the loss of nodes, as long as there is a majority of ordering nodes (referred to as a "quorum") remaining. In other words, if there are three nodes in a channel, it can withstand the loss of one node (leaving two remaining). If there are five nodes in a channel, it can lose two (leaving three remaining nodes). This number of nodes can be maintained because the resource usage of orderers is influenced by the number of channels and transactions in the network. As discussed in Chapter 5.3, the network currently operates with only three channels. Therefore, the configuration is well below the threshold where performance issues may arise.

5.2.2 Consensus Algorithm

The highly modular architecture of Hyperledger Fabric allows for integrating various consensus algorithms with the ordering service. However, implementing different consensus algorithms can often be complicated as they are not supported by default in the framework. Currently, the supported algorithms include Kafka and Raft. For this project, Raft has been chosen, which is also the recommended algorithm for version 2.5 of Hyperledger Fabric. Raft contributes signif-

icantly to meeting the consistency non-functional requirement by ensuring that all orderer nodes in the network maintain a consistent and agreed-upon sequence of transactions. This version has demonstrated superior performance and easier setup than other consensus algorithms, as explained in Section 3.

5.2.3 State Database

Hyperledger Fabric restricts the state database options for peers primarily to two choices: LevelDB and CouchDB. LevelDB serves as the default embedded key-value state database, offering basic functionality. In contrast, CouchDB provides more advanced features such as storing data in JSON format, supporting indices, and enabling complex queries. Despite a slight performance trade-off compared to LevelDB, choosing CouchDB aligns with the project's needs, emphasizing enhanced capabilities for effective data management. Section 3 investigates other state database alternatives that could be used.

5.2.4 Certificate Authority (CA)

The decision has been made to utilize a single CA server for issuing identity and TLS certificates for every MSP within the organization. This decision is based on the expectation of limited usage of the provided API by these organizations. For storing essential data such as user identities, affiliations, credentials, and public certificates, the Fabric CA server offers four options: SQLite, PostgreSQL, MySQL, and LDAP.

SQLite, being an embedded database, is unsuitable for running the CA server in a clustered environment. Therefore, the choice between PostgreSQL and MySQL is necessary to support cluster deployment. The decision to opt for PostgreSQL is driven by Sensefinity's existing utilization of this database type for other services, ensuring consistency and simplifying the overall infrastructure management. While the same instance of the PostgreSQL database is employed for every CA server, distinct accounts are associated with each. While LDAP is also a viable option, it is typically recommended when an organization already possesses an LDAP infrastructure. Creating a new LDAP solely for this purpose may not be cost-effective or practical.

5.2.5 File Storage System

In blockchain development, an efficient file system management solution is crucial for sharing network artifacts and chaincode data across nodes. The following options were considered:

- **Network File System (NFS)** [43]: Centralized solution that tracks file attributes such as security permissions, folder structures, and file sizes. It allows clients to access files from a central server, offering straightforward setup and cost-effectiveness.
- **InterPlanetary File System (IPFS)** [6]: Decentralized solution providing peer-to-peer file sharing for enhanced resilience and fault tolerance. However, it comes with higher costs and complexity, which could be challenging for integration.

NFS was chosen due to its ease of implementation and lower cost, aligning with the project's objective of minimizing expenses while effectively meeting essential requirements. Although IPFS supports decentralization, its cost and complexity led to the preference for NFS in this project.

5.3 Blockchain Network

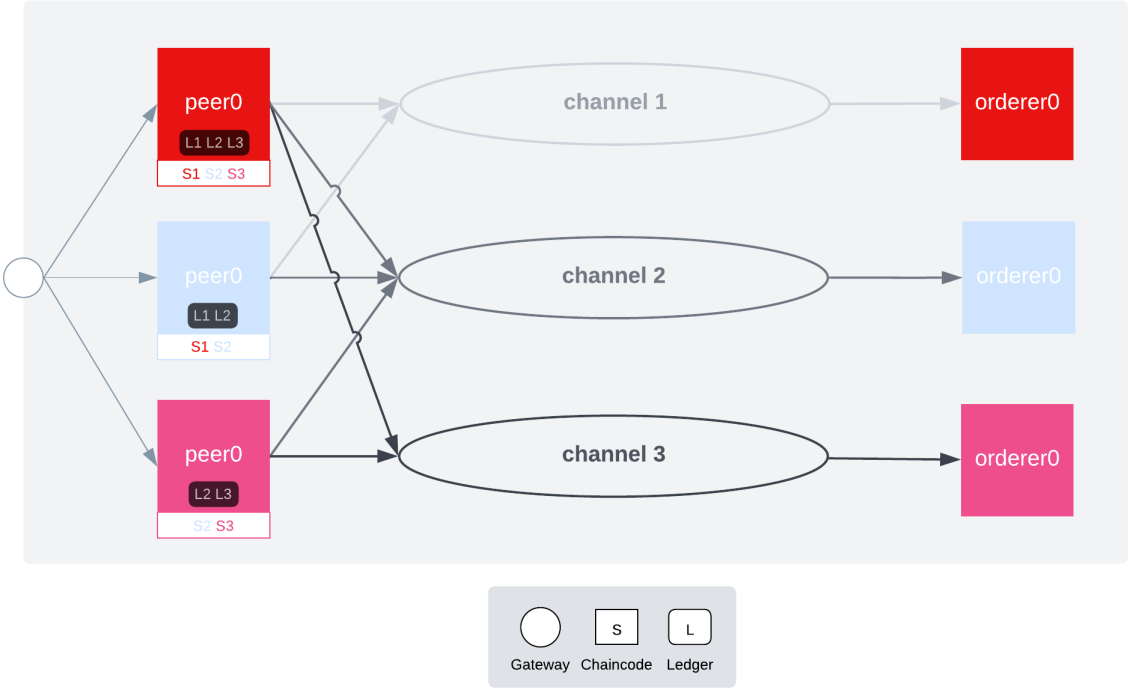

Figure 5.3: Blockchain Network

Channels play a crucial role in establishing private networks among participating organizations in Hyperledger Fabric. Their primary purpose is to maintain confidentiality, a non-functional requirement, by restricting access to transaction records within a specific group of organizations that are members of the channel. Illustrated in Figure 5.3 as channel 1, channel 2, and channel 3, these channels provide secure environments where entities like Cerfundão, Sonae, and Logistom can securely share and store sensitive information without interference from external parties. This architecture addresses functional requirements 1, 2, and 3: managing assets, retrieving asset history, and retrieving transactions, as explained below:

5.3.1 Channels

Each channel is designed to cater to specific aspects of the business, such as safeguarding data related to cherry quality, managing transportation logistics, and overseeing contractual agreements, aligning with the assets identified by the project requirements and illustrated in Appendix A.

Channel 1: Cherry Information

Channel 1 operates to maintain the confidentiality of cherry quality information exclusively between Cerfundão and Logistom. This encompasses essential details like the lot number of cherry packages, which serve as a gateway to accessing comprehensive cherry-related data, including cultivation details such as the grower's identity, pesticide, and fertilizer usage, harvest timelines, and more. Cerfundão, according to functional requirement 1, utilizes their dedicated chaincode (S1) for CRUD operations on cherry assets, while Logistom, sharing channel 1 with Cerfundão and having access to the same ledger (L1), is restricted to read-only access. This setup ensures that Cerfundão is responsible for transmitting this information whenever a batch of cherries is prepared for shipment within the network. Consequently, Logistom gains visibility into the cherries primed for transportation, enabling them to make informed decisions regarding logistics optimization.

Channel 2: Transportation Information

Channel 2 facilitates the exchange of transportation information among Cerfundão, Sonae, and Logistom. The shared information includes the location of distribution warehouses where the transporter deposits the cherries, delivery schedules, transporter identification, and most importantly, the temperature within the transporter containers. Given that cherries require precise environmental conditions during transport, monitoring the container temperature is crucial and will be covered in the next project phase. Logistom, according to functional requirement 1, utilizes its dedicated chaincode (S2) for CRUD operations on transportation assets, while Cerfundão and Sonae, sharing channel 2 with Logistom and having access to the same ledger (L2), are restricted to read-only access. This setup ensures that the transporter is responsible for transmitting this information, providing all organizations with equal visibility into critical information regarding the tracking of cherries during transport.

Channel 3: Payments Information

Channel 3 serves to facilitate the exchange of cherry requests and the storage of payment contracts between Sonae and Cerfundão. These contracts outline the number of cherries to be delivered and the associated payments, which may extend over several months as per the agreement between Cerfundão and Sonae. According to functional requirement 1, Sonae utilizes its dedicated chaincode (S3) for CRUD operations on payment contracts, while Cerfundão is restricted to read and update operations. With both organizations having access to the same ledger (L3), any discrepancies in the quantity of cherries delivered or payments are promptly identified.

Chaincode QSCC

In addition to the dedicated chaincodes for specific operations, every organization has access to QSCC (Query System Chaincode), as specified in functional requirement 3. QSCC is utilized for system-level queries on the blockchain, such as retrieving information about the blockchain state, chain information, and transaction details.

5.3.2 Gateway

API integration is crucial for accessing the blockchain, as outlined in functional requirements 1, 2, and 3. Figure 5.4 illustrates the gateway used to perform operations specified in these requirements.

In this setup, the Web Asset Management application from Sensefinity interacts with the backend system via a REST API. During this interaction, a request header is passed containing critical information such as smart contract names, channel names, peer addresses, and other necessary details required for blockchain interaction. Subsequently, the backend utilizes the Fabric-Gateway Client API [26] to establish a gRPC connection. This connection is essential for communicating with the peer nodes within the Hyperledger Fabric network, leveraging the information provided in the request header. The backend system retrieves certificates referenced in the request header from the Google Cloud Secret Manager, thereby leveraging Google Cloud for the secure storage of secrets.

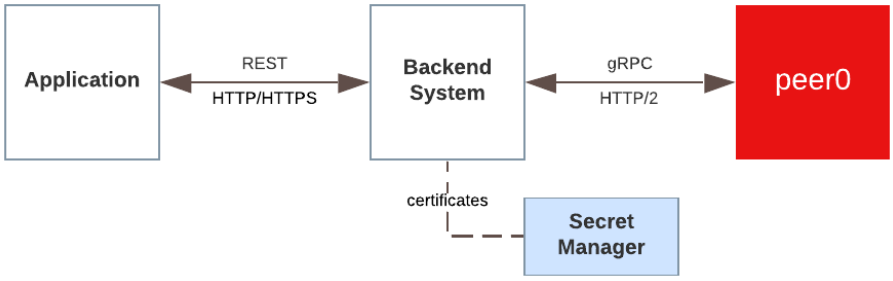

Figure 5.4: Application to Hyperledger Fabric Interaction

5.3.3 Architecture Decisions

Chaincode per Asset Type

Separating chaincodes per asset type is considered best practice because it adheres to the principle of separation of concerns. This promotes modularity, a non-functional requirement, which enhances readability and maintainability by focusing each chaincode on a single asset type. Each organization can independently manage and support its chaincode, ensuring unique requirements are met and adding a layer of defense if one chaincode is compromised. Additionally, this approach avoids the complexity and potential errors of a single, large chaincode handling various asset types. However, it does entail higher infrastructure costs.

Endorsing Peers within a Channel

Having multiple endorsing peers enhances network resilience and reliability by ensuring agreement on transaction validity among all organizations within a channel. Each organization participating in a channel acts as an endorsing peer for the specific chaincode associated with that channel (e.g., S1, S2, S3). This setup, where Cerfundão and Logistom in channel 1 use chaincode

S1, and others similarly in channels 2 and 3 use S2 and S3 respectively, simplifies the network structure and promotes transaction consistency, a non-functional requirement. It ensures that all organizations can validate and endorse transactions, thereby strengthening the integrity and reliability of the blockchain network. This approach is costly, but using chaincode as an external service can mitigate these expenses.

Orderer Node per Channel

Each channel is linked to its organization's specific orderer, meaning that each organization's orderer nodes handle transactions exclusively for channels associated with their organization. Orderer nodes from other organizations sharing the network aren't directly involved in this process. However, all orderer nodes collaborate to agree on the order of transactions across all channels, thereby ensuring consistency, a non-functional requirement, for accurate ledger updates based on the network's consensus mechanism.

5.4 Kubernetes Architecture

In Fabric deployment on Kubernetes, all components are containerized within pods, utilizing namespaces to segregate organizations and maintain isolated cluster resources, as depicted in Figure 5.5. By implementing namespaces, organizations gain controlled access to designated areas in the cloud, effectively separating and securing their data and operations. This architecture addresses the non-functional requirements of performance and meets all functional requirements by involving the deployment of the previously introduced components.

5.4.1 Deployment Strategy

Peers and CouchDBs are consolidated within single pods, adhering to best deployment practices. This setup simplifies configuration and enhances security by minimizing peer-CouchDB connection challenges. Combining these components in one pod also boosts performance by reducing network latency. Similarly, each orderer is isolated within its pod to streamline management and optimize performance.

Each organization independently manages its Certificate Authority (CA) with dedicated pods for OrdererMSP and PeerMSP. The CA connects to an external PostgreSQL database deployed by Sensefinity via its private IP, as both are within the same private network. Although the CA service can be turned off during periods of inactivity, it remains critical for handling cryptographic operations within the organization, necessitating dedicated resources to ensure consistent functionality. Additionally, the Command-Line Interface (CLI) for managing identities, specified in functional requirement 5, is omitted due to low anticipated usage. Instead, the CLI is utilized locally with the service exposed for interaction.

A Command-Line Interface (CLI) pod has been included, providing a convenient space for executing command-line tools to interact with the Hyperledger Fabric network and manage nodes

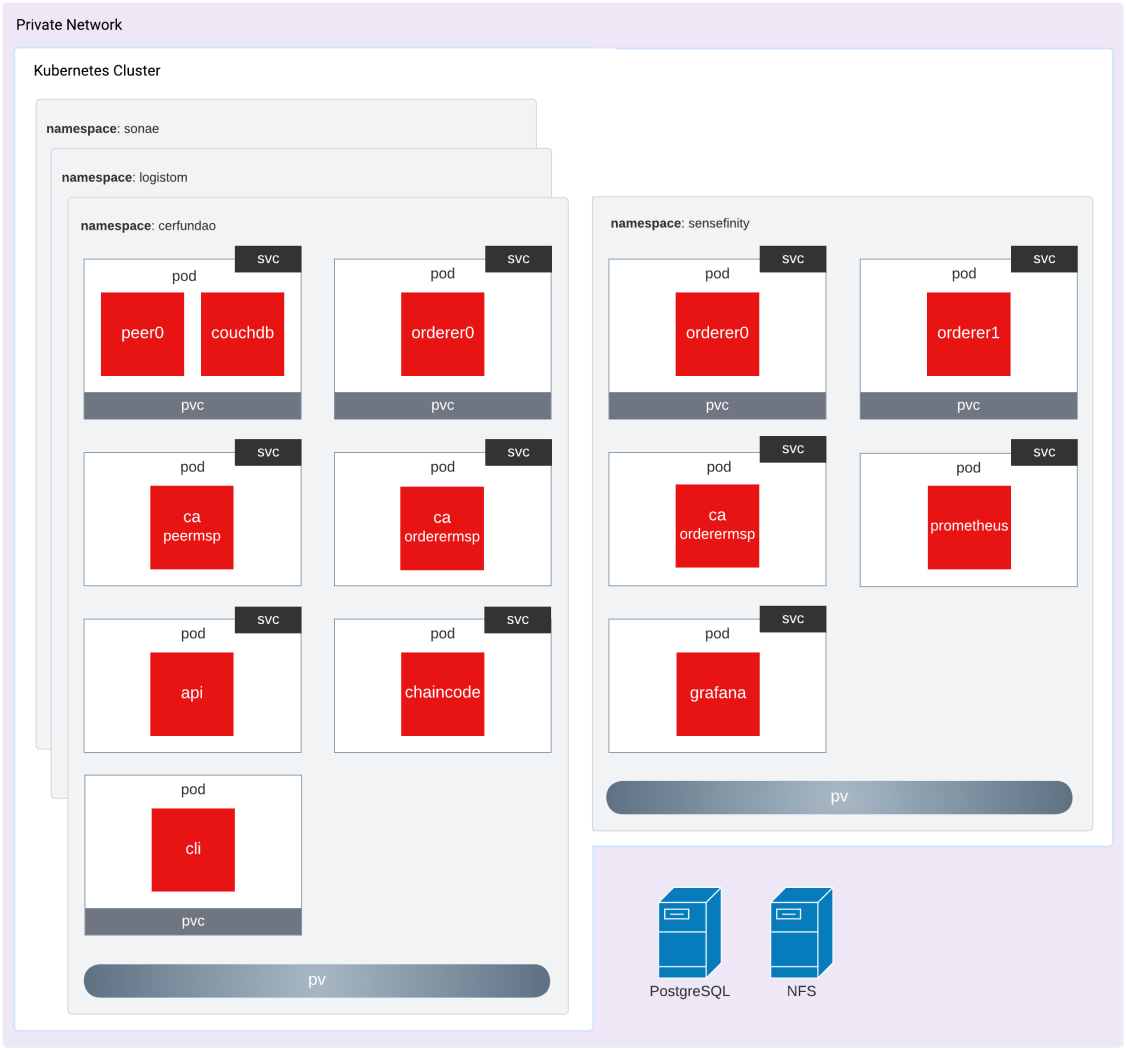

Figure 5.5: Kubernetes Architecture

within the blockchain, as per functional requirement 4. In addition, the chaincode of each organization is deployed using an external service approach. The respective API for this chaincode, which serves as a gateway to interact with the blockchain as specified in the previous chapter, is also deployed within a pod. Moreover, Prometheus has been deployed in the Sensefinity namespace to collect metrics from the peers and orderers of each organization, as required by functional requirement 6. Grafana is used to visualize this data.

Auto Scaling

The architecture employs **Vertical Pod Autoscaler (VPA)** to adjust resources (vCPUs and RAM) based on current demand. Pods begin with a baseline allocation, and the VPA dynamically updates these resources to maintain optimal performance as workloads change. This ensures efficient adaptation to varying demands without manual intervention. Additionally, **Google Cloud's Cluster Autoscaler** is used to manage the overall cluster resources by dynamically scaling the number

of nodes. This ensures that there are always sufficient resources available for the pods while optimizing costs. When demand increases, new nodes are added to the cluster; when demand decreases, the number of nodes is reduced, balancing resource availability and cost-effectiveness.

5.4.2 Chaincode as an External Service

In this architecture, chaincode is deployed as a dedicated pod within the Kubernetes cluster, offering significant advantages. This configuration allows chaincode to operate independently from individual peers, enhancing modularity and simplicity, both key non-functional requirements. Rather than deploying chaincode on each peer, it runs externally, simplifying updates. Peers only need to approve changes, avoiding the necessity of reinstalling chaincode across all peers—a time-consuming process in larger networks. This approach reduces the load on individual peers by referencing the chaincode location instead of installing entire packages, which is particularly beneficial for endorsing peers requiring multiple chaincodes. Centralizing resources in chaincode pods also enhances resource management efficiency in dynamic cloud environments.

5.4.3 Service Exposure

The Kubernetes Gateway API [37] is selected over Ingress for exposing services outside the cluster, including CA servers and the API, due to its several advantages. The HTTPRoute resource of the Gateway API allows for flexible traffic routing based on host, header, and path fields. This is illustrated in Figure 5.6, which shows how HTTPRoute resources direct traffic to various Kubernetes Services:

- Traffic to **blockchain.sensefinity.com/cerfundao/api/*** is forwarded to **api-cerfundao**.
- Traffic to **blockchain.sensefinity.com/logistom/api/*** is forwarded to **ca-logistom**.

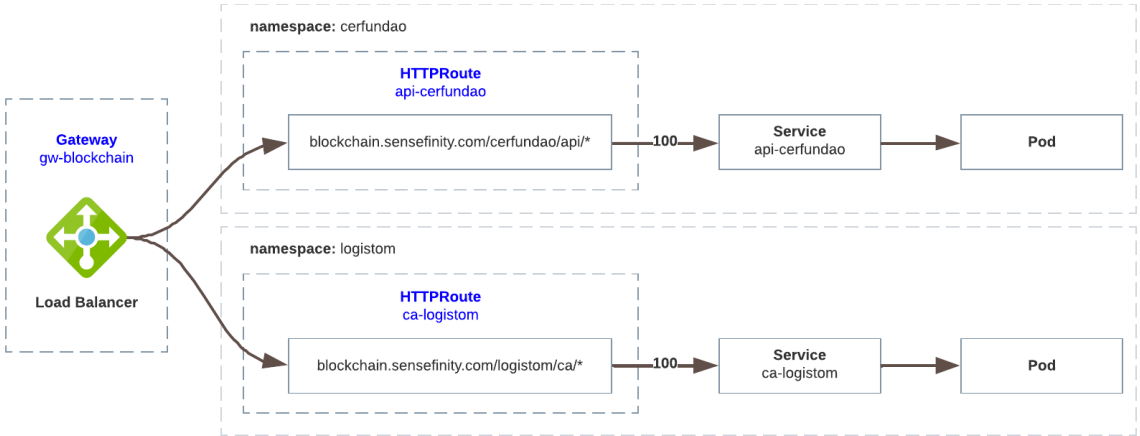

Figure 5.6: HTTP Routing with Kubernetes Gateway API

This configuration demonstrates the Gateway's capability to manage cross-namespace routing with a single load balancer, unlike Ingress, which typically requires one load balancer per

namespace, resulting in higher costs. The Gateway API simplifies the setup by keeping services as ClusterIP, thus avoiding the need for NodePort services and reducing complexity.

In the current setup, traffic is directed to the specified services without replication, so load balancing is not utilized. However, the Gateway API is designed to handle traffic distribution effectively when services are replicated. It can distribute traffic across multiple replicas of a service, supporting efficient load balancing and scalability. This ensures that during high-traffic conditions, resources can be allocated dynamically to maintain performance and manage varying load levels.

5.4.4 Data Persistence

In Hyperledger Fabric, nodes are stateless, requiring external databases or shared storage for data persistence. Each organization is allocated a Persistent Volume (PV) connected to an NFS (Network File System) via its private IP, as both are within the same private network, for storing blockchain data. NFS serves as the centralized storage, maintaining data across pod restarts or failures. PVs are mapped to specific directories on the NFS server, ensuring separation and access control between organizations. Persistent Volume Claims (PVCs) abstract this setup, allowing pods to easily access and utilize resources from PVs. This integration of NFS, PVs, and PVCs supports efficient data persistence and access management in Hyperledger Fabric, facilitating smooth operations across organizational units.

5.5 Summary

This chapter outlines the architectural solution for integrating a blockchain system into the supply chain, focusing on the ideal decentralized design, adjustments for the pilot phase, network architecture, and Kubernetes deployment. The ideal architecture proposes a fully decentralized setup where different organizations, each with its own Membership Service Provider (MSP), manage their roles as peers, orderers, and Certificate Authorities (CAs). This structure ensures high availability and fault tolerance. The pilot phase simplifies this design by deploying fewer peers and orderers and using a single CA server per organization to balance functionality with cost.

The network architecture uses distinct channels to handle cherry information, transportation logistics, and payment contracts. Channel 1 connects Cerfundão and Logistom for cherry data, Channel 2 involves Cerfundão, Sonae, and Logistom for transportation details, and Channel 3 is used by Cerfundão and Sonae for payment information. Each channel ensures privacy and secure data sharing among relevant stakeholders, with dedicated chaincodes managing asset data and transactions. Raft consensus and CouchDB are used for robust transaction ordering and advanced data management. Kubernetes supports deployment by containerizing components, facilitating auto-scaling, and managing resources. Chaincode is deployed as an external service, and the Kubernetes Gateway API handles efficient service exposure and traffic routing. The following chapters will provide detailed implementation procedures for this architectural solution.

Chapter 6

Blockchain Implementation

In this chapter, the implementation of the blockchain is detailed, following the steps outlined in Siddharth Jain's book [33]. The examples are applied to the Cerfundão organization, the central entity in the project, which manages information about cherries. The same principles are also applied to the other organizations within the consortium. The implementation uses Hyperledger Fabric version 2.5 and is organized into several sections.

- **Section 6.1**: Introduces the creation of Certificate Authorities (CAs) to provide unique identities and TLS certificates for the blockchain nodes.
- **Section 6.2**: Explains the process of provisioning nodes using the previously generated certificates.
- **Section 6.3**: Details the steps leading up to the provisioning of a channel and the installation of chaincode by an organization's administrator.
- **Section 6.4**: Presents conclusions drawn from the chapter's content.

6.1 Creating Blockchain Identities

The first step to provision a Fabric network is to generate identities for the nodes that compose the blockchain network. These identities differ from the conventional use of usernames and passwords common nowadays, as they require the use of digital certificates, specifically X.509 Certificates [13]. Listing 6.1 defines the four key components of these certificates. In its most basic form, an X.509 certificate includes:

- **Public key:** Asymmetric key cryptography utilizes two keys: a private key, which is never shared with anyone, and its corresponding public key.
- **Identity:** Specifies who owns that key.
- **Permissions:** Defines how the keys can be used, typically outlined by permissions.
- **Issuer:** Indicates the entity that issued the certificate.

```
type X509 struct {
    PublicKey  []byte
    KeyUsages  int64
    Metadata   string
    Signature  []byte
    Issuer     *X509
}
```

Listing 6.1: X509 Struct

To establish identities for the organizations and their components, including human users and administrators as well as machine entities such as peers and orderers, the process begins with deploying a certificate server for each Membership Service Provider (MSP) within every organization. This server automatically creates the CA root certificate that will issue all other certificates. The preferred tool for generating these certificates is the fabric-ca-server provided by Hyperledger Fabric. This server automatically produces certificates in the MSP structure, which aligns perfectly with the authentication requirements of Fabric nodes. This makes it the favored choice over alternative options. The configuration of these certificate servers is facilitated using the YAML configuration files [24] provided by Hyperledger Fabric. Once configured, these servers are transformed into Docker images and deployed on Kubernetes (see Appendix C.1). After deploying the certificate servers, the Fabric-provided CLI [23] is configured to interact with these servers to generate the required identities. Overall, the flow of creating a certificate is illustrated in Figure 6.1. The CLI interacts with the server, which in turn creates the certificates in the MSP format and stores data in the database.

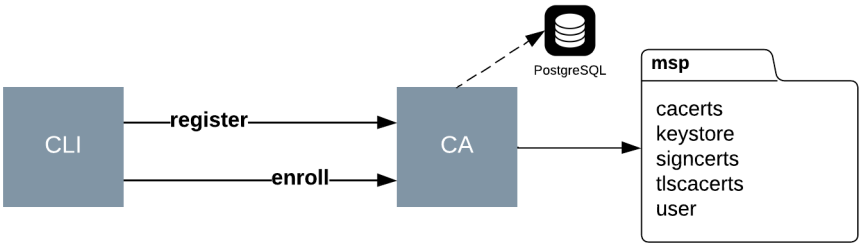

Figure 6.1: CA MSP visualization

The first step in this process is to enroll the bootstrap user (Listing 6.2), who has the authority to enroll other users and create certificates within the system. The credentials of this bootstrap user are typically provided in the configuration file of the server.

```
fabric-ca-client enroll
-u "https://sensefinity:secret@$localhost:3000"
```

Listing 6.2: Enroll Bootstrap User

The command referenced in Listing 6.3 registers a Sensefinity orderer node. The **-u** flag is utilized to specify the URL of the Fabric CA server. The **–id.name** flag sets the username of the user being registered, while **–id.type** determines the user's type, which will be reflected as **OU=orderer** in the user's X.509 certificate. This designation signifies that the certificates issued to this user have orderer permissions, which are defined within the blockchain through policies.

Depending on this type, the user can be assigned various roles such as peer, orderer, or client. These roles are registered on the blockchain, allowing it to verify permissions associated with each role.

```
fabric-ca-client register
--id.name orderer0-sensefinity
--id.secret secret
--id.type orderer
```

Listing 6.3: Register Orderer

The register command essentially creates a user account in the CA's server database, but it does not generate the certificates. To create the certificates, the user needs to be enrolled, as referenced in Listing 6.4. The **-M** flag is used to specify an output directory for the certificates in the enroll command. Both public and private keys are generated on the client side. This is most important for private keys since the private key is the client's property and should never be exposed to anyone.

```
fabric-ca-client enroll
-u "https://orderer0-sensefinity:secret@localhost:3000"
-M sensefinity/orderers/orderer0/msp
```

Listing 6.4: Enroll Orderer

To run the orderer with TLS enabled, TLS certificates need to be created, as referenced in Listing 6.5. Essentially, it involves using the same enroll command as before, but with an additional flag, **–enrollment.profile**, to indicate an enrollment of type TLS. Additionally, the **–csr.hosts** flag is used to specify all addresses through which the orderer can be accessed. These hosts will appear in the Subject Alternative Name (SAN) section of the certificate. For instance, the host orderer0-sensefinity.sensefinity.svc.cluster.local is the Fully Qualified Domain Name (FQDN) of the orderer. This allows other pods in Kubernetes to communicate with the orderer across different namespaces within the Kubernetes cluster.

```
fabric-ca-client enroll
-u "https://orderer0-sensefinity:secret@localhost:3000"
--enrollment.profile tls
--csr.hosts orderer0-sensefinity
--csr.hosts localhost
--csr.hosts orderer0-sensefinity.sensefinity.svc.cluster.local
-M sensefinity/orderers/orderer0/tls/msp
```

Listing 6.5: Enroll Orderer TLS

The same logic is applied to creating the certificates for peers, users, and admins. However, it's important to note that users and admins do not need to undergo the final step, as TLS certificate enrollment is specifically intended for machines.

6.2 Provisioning Blockchain Network

In the previous chapter, the necessary identities to bootstrap peer and orderer nodes were generated. Utilizing these identities, the following steps in this chapter demonstrate how the network designed in the system architecture was built. The steps are as follows:

1. Generate the Genesis Block
2. Provision the Orderer Node
3. Provision CouchDB
4. Provision Peer Nodes
5. Provisioning Prometheus and Grafana

6.2.1 Generating Genesis Block

The genesis block is the initial block of the blockchain and is used to initialize the ordering service. It contains the public certificates generated earlier and includes the necessary information for runtime requests to Fabric. This enables Fabric to determine the organization to which the caller belongs and the privileges the caller has. Additionally, the genesis block contains details regarding the provisioning of orderers and the addresses of peer nodes used for connecting to the orderer. All this information is outlined in the configtx.yaml configuration file [25], which acts as a comprehensive blueprint for configuring the blockchain network.

The genesis block is generated by a utility known as **configtxgen**, one of the binaries provided by Hyperledger Fabric to create these artifacts. Listing 6.6 shows the script used to generate the genesis block using this binary. The flag **-configPath** is used to specify the location of the configtx.yaml file, the **-profile** flag specifies the type of artifact to create, the **-outputBlock** flag specifies the name of the block, and the **-channelID** flag specifies the name of the channel. This channel is of the type system channel, where consortium configuration is stored.

```
configtxgen -configPath ./channel-artifacts -profile Genesis -outputBlock
    genesis.block -channelID system
```

Listing 6.6: Generate Genesis Block

6.2.2 Provisioning Orderer Nodes

Genesis block created beforehand, an ordering node can be bootstrapped. The configuration file used to configure the orderer is orderer.yaml [27]. This file contains the configuration settings that the orderer will read when it's launched, including the location of the genesis block. When this orderer is transformed into a Docker image for future deployment (see Appendix C.2), the Docker image needs to run using the instruction **CMD ["orderer"]**, so that it executes the command provided by the orderer binary. This configuration file can be divided into several sections, as described in Table 6.1, to facilitate a better understanding of the file.

6.2.3 Provisioning CouchDB

The state database provisioning for each peer is facilitated by the image **couchdb:3.3.3**. This image is incorporated into the deployment file within Kubernetes, residing in the same pod as the peer to streamline the deployment process. CouchDB proves particularly useful, especially during

Section	Purpose
General	Configures general settings such as where the orderer will listen for connections, which certificates will be used for TLS, etc.
FileLedger	Specifies the location where the orderer will store its data.
Debug	Enables verbose tracing of broadcast and delivery requests to the orderer.
Operations	Configures the operations server endpoint for the orderer.
Metrics	Configures metrics collection for the orderer (e.g., Prometheus).
Consensus	Specifies the directories for write-ahead logs and snapshots used by the Raft algorithm.

Table 6.1: Orderer Configurations

the development phase, due to the user interface it offers for viewing and inspecting the world state.

6.2.4 Provisioning Peer Nodes

This chapter focuses on provisioning the peer nodes for each organization. Similar to provisioning the orderer, the peer has a configuration file named core.yaml [28] that contains configuration settings that the peer will read when it's executed. For the deployment of the peer (see Appendix C.3), the Docker image needs to run using the instruction **CMD ["peer", "node", "start"]**, so that it executes the command provided by the peer binary. This configuration file is comprised of the following sections, as depicted in Table 6.2.

Section	Purpose
Peer	Configures general settings such as where the peer will listen for connections and which certificates will be used for TLS.
VM	Provides connection string to the Docker daemon necessary when the peer needs to spin up a chaincode container.
Chaincode	Determines the mode in which the peer is running and provides various settings needed, such as specifying the Docker image to use for building the chaincode.
Ledger	Specifies whether to use LevelDB or CouchDB, along with other ledger-related settings.
Operations	Configures the operations server endpoint for the peer.
Metrics	Configures metrics collection for the peer, for example, using Prometheus.

Table 6.2: Peer Configurations

The chaincode section is where the external builder is defined, enabling the execution of chain-

code as an external service. This approach streamlines the deployment of chaincode on Fabric cloud deployments, such as Kubernetes. Rather than building and launching the chaincode on every peer, the chaincode can be operated as a service external to Fabric. This requires custom packaging of the peer image used in the Dockerfile for deployment, as the current official fabric-peer image does not include necessary samples such as **jq** and **bash**.

The first step is to modify the chaincode section of the peer's core.yaml file to include the externalBuilders configuration element, as shown in Listing 6.7.

```
externalBuilders:
     - name: external-builder
       path: /home/external-builder
```

Listing 6.7: External Builder

The path specifies the locations of each script, including the **detect**, **build**, and **release** file scripts. Each of these scripts is responsible for:

- **detect:** Determines whether a Buildpack should be used to build the chaincode package and launch it.
- **build:** Transforms the chaincode package into executable chaincode. In simpler terms, this script is responsible for building, compiling, or transforming the contents of a chaincode package into artifacts that can be used for release and execution.
- **release:** Provides chaincode metadata to the peer.

6.2.5 Provisioning Prometheus and Grafana

For monitoring activity in each blockchain node, the Prometheus provider is included in the metrics section of each node's configuration file, as detailed in Listing 6.8. Metrics are exposed on the port specified in the operations section and can be accessed through the **/metrics** endpoint.

```
metrics:
    provider: prometheus
```

Listing 6.8: Prometheus Provider

Prometheus and Grafana are both deployed using their official images. The Prometheus server is configured to scrape metrics from the endpoints specified for each node's operation server. Grafana then retrieves these metrics from Prometheus and displays them using a predefined template, as illustrated in Figure 6.2.

6.3 Provisioning a Channel and Installing the Chaincode

The chapter covers the remaining steps required before requests can be made to perform transactions on the blockchain, following the provisioning of the network. These steps are as follows:

1. Setting Up the Admin Environment

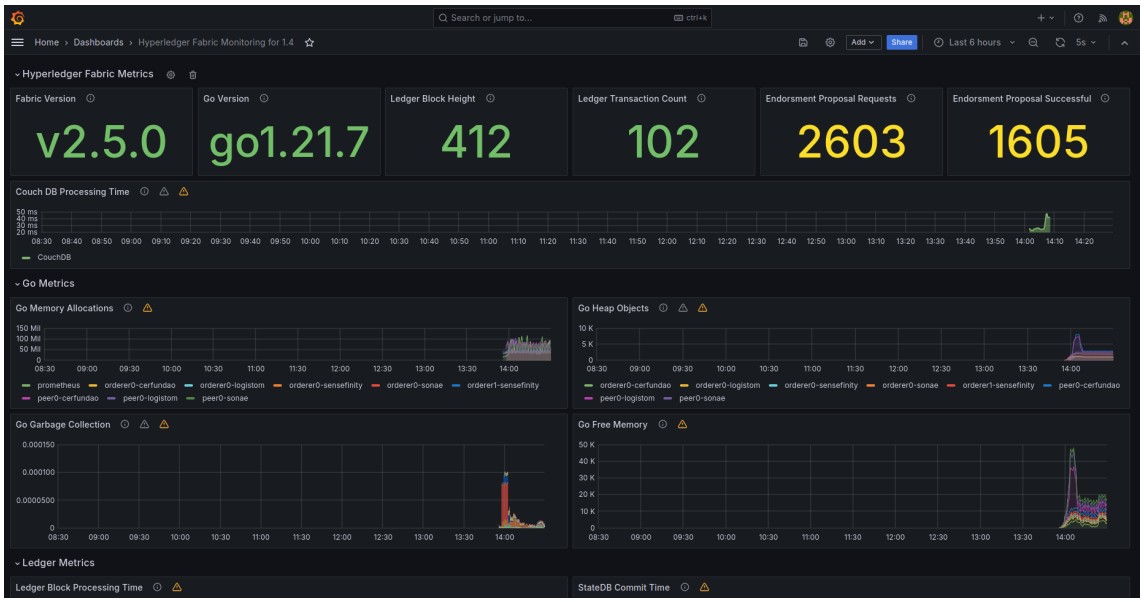

Figure 6.2: Hyperledger Fabric Monitoring With Grafana

2. Create a Channel
3. Join Peer Nodes to the Channel
4. Install Chaincode on Peer Nodes
5. Approve and Commit Chaincode on the Channel
6. Update Anchor Peers

6.3.1 Setting Up the Admin Environment

Only administrators are allowed to perform the aforementioned steps. Therefore, a peer CLI needs to be deployed to interact with the network and execute commands. The peer binary serves a dual purpose: in the previous chapter, it was used to start the peer, but in this one, it is utilized for administrator functions such as creating channels, joining peer nodes to the channel, and installing, instantiating, or upgrading chaincodes. These functions are available in the image **hyperledger/fabric-tools:2.5**, so a container is deployed with this image (see Appendix C.4). The CLI is configured through a core.yaml file, which is much shorter than the previous one used for running the peer, as it only requires the peer section and a few settings as listed in Listing 6.9.

```
peer:
    address: peer0-cerfundao:8050
    mspConfigPath: msp
    localMspId: CerfundaoPeerMSP
    localMspType: bccsp
    tls:
        enabled:  true
        clientAuthRequired: true
    keepalive: ...
    client: ...
```

```
    BCCSP: ...
```

Listing 6.9: Peer CLI Configuration

The explanations are as follows: **address** is used to specify the address of the server that the client will connect to. **mspConfigPath** contains the identity that will be used to bootstrap the peer client. **localMspID** is the MSP ID of the user's organization. The rest of the sections were copied from the core.yaml file used for peer execution, as they remain the same.

6.3.2 Create a Channel

A channel is established through a channel configuration update transaction. Specifically, this transaction is intended to create a configuration for the new channel and is processed by a specialized system chaincode residing on the ordering service. Figure 6.3 is a flowchart outlining the steps required to create a channel in the blockchain network. These steps consist of two stages:

- Creating a channel configuration transaction (channel.tx)
- Utilizing the channel configuration transaction to establish the channel

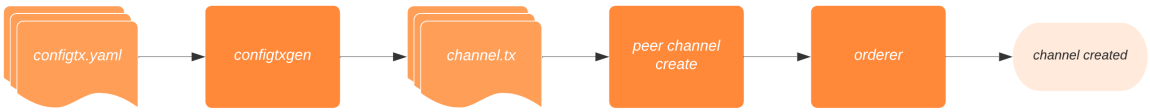

Figure 6.3: Channel Creation Flowchart

The configuration file (**channel.tx**) is once again generated using the **configtxgen** tool to interpret the **configtx.yaml** configuration file, as depicted in Listing 6.10. After generating this configuration file, it needs to be copied to the CLI container so that the admin can utilize it to generate the channel through this transaction file. This is accomplished by mounting a directory in the CLI container to the NFS server, enabling these artifacts to be easily transferred between environments.

```
configtxgen -configPath ./channel-artifacts -profile Channel -
    outputCreateChannelTx cherriesquality.tx -channelID cherriesquality
```

Listing 6.10: Channel Configuration Transaction

With the channel configuration transaction inside the CLI container, the administrator from an organization peer can execute the command using the peer binary, as shown in listing 6.11.

```
peer channel create -o orderer0-cerfundao.cerfundao.svc.cluster.local:7052 -c
    cherriesquality -f ./artifacts/cherriesquality.tx --outputBlock ./artifacts
    /cherriesquality.block \
--tls \
--cafile ./certs/cerfundao/orderermsp/orderers/orderer0/tls/ca.crt \
--clientauth \
--certfile ./certs/cerfundao/orderermsp/orderers/orderer0/tls/server.crt \
--keyfile ./certs/cerfundao/orderermsp/orderers/orderer0/tls/server.key
```

Listing 6.11: Create Channel

The **peer channel create** command submits a request to the orderer to create a channel, defined by the **-o** flag, which contains the address of the Cerfundão orderer. This request, along with all peer commands, depends on the variables configured in the core.yaml configuration file, specifically the **localMspId** and **mspConfigPath** variables from the peer section. The **localMspId** variable specifies the identity of the requester, while the **mspConfigPath** variable utilizes the provided identity from the public certificate. Therefore, when the request reaches the orderer at the provided address, it can verify that the caller is an admin of Cerfundão.

Every node in the network employs mutual TLS. With mutual TLS, not only does a node send a TLS certificate to a client, but the client also needs to provide a certificate to authenticate to the node. This effectively acts as a whitelist, allowing only authorized entities to connect to nodes. As a result of this setup, whenever an administrator makes a request that necessitates access to an orderer, certain flags for TLS authentication must be specified. The **–tls** flag is utilized to denote that the orderer is operating with TLS enabled, prompting the client to conduct a TLS Handshake. Additionally, the **–cafile** flag contains the certificate of the CA, which is once again used to validate the orderer certificate. If the orderer is running with two-way mutual TLS, the **–clientauth** flag must be set. In this case, the **–certfile** and **–keyfile** flags are utilized to provide the certificate and key that the client will use to authenticate itself to the server.

6.3.3 Join Peers to Channel

To enable peer nodes to join the channel, administrators from each peer organization must follow a three-step process:

1. Fetch the Genesis Block of the Application Channel
2. Verify the Correctness of the Genesis Block
3. Join the Channel Using the Fetched Block

In a real-world environment, adherence to steps 1 and 2 is crucial. In step 1, the administrator from a peer node fetches the genesis block from the application channel using the command specified in Listing 6.12.

```
peer channel fetch 0 cherriesquality.block -o orderer0-cerfundao.cerfundao.svc.
    cluster.local -c cherriesquality
```

Listing 6.12: Fetch Genesis Block

Step 2 involves verifying various aspects, including policies, certificates, and other pertinent information, to ensure they align with the administrator's expectations before proceeding to join the channel. This verification process can be achieved by utilizing the **configtxlator** binary to decode the block and inspect the associated policies.

```
configtxlator proto_decode --type common.Block --input cherriesquality.block
```

Listing 6.13: Decode Block

It's essential to note that in this scenario, the development process seemingly skipped steps 1 and 2 and proceeded directly to step 3, utilizing the command outlined in listing 3 to join the channel using the fetched block.

```
peer channel join -b cherriesquality.block
```

Listing 6.14: Join Channel

The command above establishes the client's connection to the target peer. Subsequently, the peer connects to the orderer. However, the orderer address is not explicitly specified in the command to join the peer to a channel. Instead, the orderer address is derived from the connection information provided to the peer from the cherriesquality.block passed to it. If the channel contains existing data, the peer will initiate the process of downloading blocks from the orderer and other peers using the gossip protocol. This mechanism enables new organizations joining a consortium to obtain a copy of the existing blockchain and historical data.

6.3.4 Install Chaincode on Peer Nodes

This chapter marks the stage where the chaincode is installed on the peer nodes. Essentially, it involves a two-step process:

1. Packaging the Chaincode into a Tar File
2. Installing the Packaged Chaincode onto the Peer Nodes

When the chaincode is created in Chapter 7, a packaging folder is generated containing two essential files: connection.json and metadata.json. These files hold configuration information about the chaincode, which is later utilized by the external builder and launcher process, as referenced in Section 6.2.4. The **detect** script determines if the chaincode is an external service using the metadata.json file, while the connection.json file provides the chaincode endpoint information used by the **build** script and is supplied to the peer by the **release** script. This means that only these files need to be packaged, not the chaincode itself, as shown in listing 6.15.

```
tar cfz code.tar.gz connection.json
tar cfz cherriesqualitycc.tgz code.tar.gz metadata.json
```

Listing 6.15: Package Chaincode

After the chaincode is packaged, it can be installed on the peer nodes by the administrator of an organization using the CLI container. The command specified in Listing 6.16 is used for this purpose.

```
peer lifecycle chaincode install cherriesqualitycc.tgz
```

Listing 6.16: Install Chaincode

Normally, this command takes a while since the process of installing chaincode on a peer involves the building of a Docker image, which is not required when using chaincode as an external service. In this case, the chaincode is already built in another container, making this step almost instant.

The output of this command will generate a chaincode package identifier, which serves as a tag to ensure all organizations synchronize to the same snapshot of the code and avoid errors downstream resulting from using different versions of the chaincode packages. This chaincode package identifier is also used by administrators to approve and commit chaincodes, as well as to configure the server where the chaincode runs.

6.3.5 Approve and Commit Chaincode on the Channel

After the chaincode is installed, approval for its usage needs to be granted by each administrator from a peer organization. This is done using the command specified in Listing 6.17.

```
peer lifecycle chaincode approveformyorg
-o orderer0-cerfundao.cerfundao.svc.cluster.local:7052
--channelID cherriesquality
--name cherriesqualitycc
--version 1.0
--package-id cherriesqualitycc:4547852
    b21e4c27dcbc719861b82a69e18e86f8c07d89c228032b90ff66ee3c9
--sequence 1
--signature-policy "AND ('CerfundaoPeerMSP.peer', 'LogistomPeerMSP.peer')"
--waitForEvent
```

Listing 6.17: Approve Chaincode

The command above executes what is called a system chaincode transaction. Since the command is executed from the administrator CLI container, it connects to the target peer specified in the core.yaml configuration file. After establishing this connection, it obtains an endorsement from the peer itself, and this endorsement is then sent to the orderer address specified in the **-o** flag. As a result, this approval is recorded on the ledger of all organizations, indicating that one organization, in this case, Cerfundão, has approved the chaincode with a package identifier specified in the **–package-id** flag. Every time an organization wants to approve a definition, certain parameters need to be consistent between them all, namely:

- **Name:** This is the name that applications use to call the chaincode.
- **Version:** Represents a number or text associated with a specific chaincode package. When the chaincode is updated, the version should be changed.
- **Sequence:** Refers to a number assigned to the chaincode when it's defined. It helps track chaincode updates.
- **Signature Policies:** These are the rules governing the endorsement of transactions. In the provided listing, the **–signature-policy** flag indicates that both organizations must endorse a transaction before it can be added to the ledger. If not explicitly specified, Fabric resorts to the default endorsement policy outlined in the configtx.yaml file, typically relying on majority endorsement.

The **–waitForEvent** flag causes the command to block until it receives a signal from the target peer notifying the status of the approve request. The TLS section in both the approve and now commit commands has been hidden for better clarity of the command.

The last step to make the chaincode usable is to commit it. Before committing, the majority of organizations in the network need to approve. This is a policy defined in the configtx.yaml configuration file under the **Application.Policies.LifecycleEndorsement** section, as illustrated in Listing 6.18.

```
LifecycleEndorsement:
            Type: ImplicitMeta
            Rule: "MAJORITY Endorsement"
```

Listing 6.18: Lifecycle Endorsement

With three application organizations in the network (bearing in mind that Sensefinity only contributes with orderers), approval from two organizations is sufficient to satisfy this policy. Following this, the commitment of the chaincode can be accomplished by a single organization administrator, as indicated in Listing 6.19. This command entails the same consistent parameters as the approve command.

```
peer lifecycle chaincode commit
-o orderer0-cerfundao.cerfundao.svc.cluster.local:7052
--channelID cherriesquality
--name cherriesqualitycc
--version 1.0
--sequence 1
--signature-policy "AND ('CerfundaoPeerMSP.peer', 'LogistomPeerMSP.peer')"
--peerAddresses peer0-cerfundao.cerfundao.svc.cluster.local:8050
--peerAddresses peer0-logistom.logistom.svc.cluster.local:8052
--waitForEvent
```

Listing 6.19: Commit Chaincode

The network is now ready to perform transactions; however, it is only possible to use the peer CLI to invoke or query the chaincode. The goal is to perform transactions using a web application, and for that reason, it's necessary to define anchor peers in the network, as explained in the next section.

6.3.6 Update Anchor Peers

Anchor peers act as gateway nodes within an organization. The peers in Cerfundão will connect to the anchor peers of Logistom to communicate with the peers of this organization. Thus, anchor peers are critical for cross-organization communication. Without them, no cross-org communication can take place. The steps to update an anchor peer (which essentially involves modifying its role rather than creating it from scratch, as the peer already exists) include adjusting its configuration to include the role of an anchor. This step consists of two stages:

1. Generate a Transaction to Update the Anchor Peer

2. Perform a Channel Update Transaction to Update the Anchor Peer

In the first step, similar to previous implementations, an administrator creates a channel configuration transaction using the **configtxgen** binary, as shown in Listing 6.20. This step must be repeated for every channel the organization has joined, ensuring that peers within each channel can discover this anchor peer.

```
configtxgen -configPath ./artifacts -profile Channel -outputAnchorPeersUpdate
    ./artifacts/anchor0-cerfundao-cherriesquality.tx" --channelID
    cherriesquality -asOrg CerfundaoPeer
```

Listing 6.20: Anchor Peer Configuration Transaction

In the second and final step, the previously generated transaction file is utilized to execute a channel update transaction. The administrator employs the command depicted in Listing 6.21 within the CLI container to accomplish this task. For brevity, the TLS flags are omitted.

```
peer channel update -o orderer0-cerfundao.cerfundao.svc.cluster.local:7052 -c
    cherriesquality -f aux/artifacts/anchor0-cerfundao-cherriesquality.tx
```

Listing 6.21: Create Anchor Peer

The setup is now complete, and the peers in the network can be discovered, enabling them to conduct transactions. The upcoming chapters will detail the smart contracts and their respective APIs for interacting with the network, as well as the web application for visualizing blockchain data.

6.4 Summary

Implementing the blockchain infrastructure and configuring the Fabric network establish the foundation for all functional requirements. The installation of the chaincode addresses functional requirements 1, 2, and 3, which involve managing assets, retrieving asset history, and retrieving transactions. These smart contracts and their respective APIs will be explained in detail in the next chapter, Chapter 7. Additionally, the CLI deployed for Fabric network interactions satisfies functional requirement 4 for managing network operations, while the CLI for the Fabric CA server meets functional requirement 5 for managing identities. Furthermore, Prometheus collects and displays metrics from each node on a Grafana dashboard, thereby satisfying functional requirement 6 for monitoring activity.

All non-functional requirements related to security are met in this system. Blockchain technology inherently ensures data integrity. Access control is managed through the identities and roles created in the Fabric CA server. Confidentiality is maintained through the use of channels, and secure communication is upheld with TLS implemented across all nodes.

The non-functional requirements related to performance, specifically scalability and availability, are addressed by the Kubernetes deployment, as detailed in Appendix C. Kubernetes' autoscaling capabilities for pods and nodes within the Google Cloud cluster ensure that the system can scale to handle increased loads. Availability is further supported by Kubernetes' automatic failover, rolling updates, and built-in load balancing, which together maintain high availability with minimal downtime.

To address non-functional requirements for resilience, data consistency is maintained through the deployment of endorsing peers and the use of the Raft consensus mechanism in the orderers. Organizations that share the same channel ensure consistent data across their peers. Regarding fault tolerance, an optimal solution is discussed in Section 5.1, although it would be more costly. Instead, a more cost-effective solution has been implemented, as detailed in Section 5.2. Resilience is achieved through the inherent modularity of Hyperledger Fabric, ensuring that a failure in one node does not affect the operation of others. This modularity also aligns with the non-functional requirement for maintainability. Although explained in this report, detailed documentation is available on Confluence [5], where other Sensefinity documentation is also hosted.

Chapter 7

Chaincode and API Implementation

In this chapter, the process of accessing and inserting blockchain data through a web application is thoroughly explored. This includes a detailed explanation of how smart contracts interact with APIs, facilitating seamless integration with the application layer. The chapter is structured as follows:

- **Section 7.1**: Introduces smart contract development for various asset types, using the fabric-contract-api-go library.
- **Section 7.2**: Outlines the creation of chaincode-specific APIs, facilitating interaction with smart contracts while emphasizing the use of request headers for essential blockchain access variables.
- **Section 7.3**: Showcases the integration of blockchain functionalities into the Web Asset Management application, highlighting the developed pages.
- **Section 7.4**: Provides a summary of this chapter.

7.1 Creating the Chaincode

Smart contracts encapsulate the business or application logic within a blockchain network. Hyperledger Fabric provides APIs implemented in Go, Java, and JavaScript, which developers can use to create these smart contracts [31]. The decision to use Golang for developing the smart contracts in this project is influenced by its superior performance and suitability for cloud-native environments. To facilitate the development of smart contracts in Golang, the fabric-contract-api-go library is used. This API provides the necessary tools and abstractions for writing and running smart contracts on the Hyperledger Fabric platform, enabling developers to focus on the business logic while the API handles the underlying blockchain interactions.

As identified in Section 4, the blockchain tracks three types of assets: information about the cherries, transportation logistics, and retailer auditing. For each of these assets, five smart contracts are implemented to manage their lifecycle on the network, along with one auxiliary contract:

1. **Create:** This smart contract enables a specific organization to create an asset within the system. It includes security measures that ensure only authorized entities can initiate asset records.

2. **Update:** This contract facilitates updates to an asset's information, ensuring that the data remains current and accurate. It is designed to prevent unauthorized alterations.

3. **Read:** This smart contract allows users to access an asset's information by using its ID. It provides a snapshot of the asset's current state.

4. **Read All:** This contract retrieves all transactions associated with an asset recorded on the blockchain. It is primarily utilized in the application developed in Section 7.3.1 for listing purposes.

5. **Delete:** This smart contract permits the deletion of an asset from the blockchain's world-state based on its ID, ensuring that only authorized modifications are allowed.

6. **Exists:** This is an auxiliary smart contract used to check if an asset exists on the blockchain before any operation is performed.

The create, update, and delete operations operations will result in writes on the blockchain, while read operations will solely perform reads on the ledger. It's important to note that currently, there isn't a method to trace the provenance or history of an asset, although this will be addressed in the following section for future implementation. Additionally, a smart contract belonging to a special chaincode already built into the peer, called **qscc**, is utilized. This chaincode specifies contracts that contain more specific types of functions related to the blockchain. For example, given a transaction ID, it returns the block to which it belongs, along with its hash and other technical information. This functionality is utilized in the application discussed in Section 7.3.1.

In the next subsections, the create and read smart contracts are explained, showing the two operations on the blockchain (write and read). The process is the same for all other smart contracts, which are listed in Appendix D.

7.1.1 Smart Contract: Create

Below is the pseudocode (Listing 7.1) for the creation smart contract regarding the cherry asset. Certain functions included are not strictly pseudocode, as they originate from the fabric-contract-api-go library. These functions will be explained immediately following the example, as they are crucial for understanding the implementation.

```
function CreateCherry(ctx, cherryRaw):
    cherry = new Cherry
    err = parse cherryRaw into cherry
    if err:
        return err

    mspID = ctx.GetClientIdentity().GetMSPID()
    if err:
```

```
        return err
    if mspID != "CerfundaoPeerMSP":
        return error "insufficient permissions"

    txId = ctx.GetStub().GetTxID()
    cherry.TransactionID = txId

    clientID = ctx.GetClientIdentity().GetID()
    if err:
        return err

    exists = s.CherryExists(ctx, cherry.ID)
    if err:
        return err
    if exists:
        return error "the cherry {cherry.ID} already exists"

    cherry.CreatedBy = clientID
    cherry.LastModifiedBy = clientID

    cherryJSON = serialize cherry to JSON
    if err:
        return err

    err = ctx.GetStub().PutState(cherry.ID, cherryJSON)
    if err:
        return err
```

Listing 7.1: Pseudocode for Creating an Asset in a Smart Contract

The first argument of every smart contract function is the **Context** object. It consists of two essential fields:

1. **clientIdentity:** This field contains the identity of the caller, identified by their X.509 certificates.

2. **stub:** The stub object provides methods to read and update values in the LevelDB or CouchDB.

Following the **Context** object, there's an argument containing the **cherryRaw** parameter, signifying the raw input data received by the smart contract. This parameter is parsed and converted into a structured format, the Cherry object.

During the execution of the method, it retrieves the identity of the caller using **GetClientIdentity().GetID()**, revealing the Subject and Issuer attributes of the caller's X.509 certificate. This information is utilized to track who created and last modified an asset. Additionally, the method fetches the transaction ID of the current transaction using **GetStub().GetTxID()**. This transaction ID is then recorded in the database and later used to access detailed blockchain-related information via the special chaincode **qscc**. The method performs the following checks:

- It validates if the caller has permission to create the asset. For instance, only "Cerfundão" is authorized to create a cherry. This check is conducted by obtaining the MSP ID of the caller from the clientIdentity field using the **GetClientIdentity().GetMSPID()** method.

- It verifies if the asset has been previously created by executing the auxiliary smart contract **exists()**.

If these checks pass, the method proceeds to create the asset in the system. The Cherry object is then serialized into JSON format. It's crucial to serialize the asset into a JSON object to leverage the CouchDB query API, which facilitates queries against the objects stored in the database. This serialized data is subsequently used to update the ledger state with the modified Cherry object. For this purpose, the standard API call **GetStub().PutState(id, cherryJSON)** is employed to store a key-value pair in the database.

7.1.2 Smart Contract: Read

The read method(Listing 7.2) looks like the following in pseudocode:

```
function ReadCherry(ctx, id):
    cherryJSON = ctx.GetStub().GetState(id)
    if err:
        return nil, "failed to read from world state: {err}"

    if cherryJSON is nil:
        return nil, "the cherry {id} does not exist"

    cherry = new Cherry
    err = parse cherryJSON into cherry
    if err:
        return nil, err

    return cherry, nil
```

Listing 7.2: Pseudocode for Reading an Asset in a Smart Contract

The primary difference between this read smart contract and the create smart contract is that the stub field from the Context object is used to execute the function **ctx.GetStub().GetState(id)**. As the name implies, this function retrieves the state of the object based on the provided ID.

In this read smart contract, there are no checks to verify if the caller has permission to read this asset. Such permission controls are managed by the channel, as implemented in the previous chapter. This smart contract retrieves the asset in JSON format from the database and then parses it into the Cherry object for return. This process is the reverse of the create smart contract, where the Cherry object is first constructed and then serialized into JSON format before being stored.

To read the asset's history, including modified versions, the function **ctx.GetStub().GetHistory ForKey(key)** is used. This function returns a list of assets, which must be iterated through. However, a specific smart contract for handling this process was not implemented.

7.2 Creating the API

Hyperledger Fabric provides the Fabric Gateway client API [31], a minimal API for submitting transactions to a Fabric network, with support for Go, Java, and JavaScript. All transaction proposals to record or query information on the blockchain pass through this layer. It is a good practice

to abstract this process behind an intermediate layer of communication. In this work, a REST API serves as the abstraction, aligning with the existing system architecture in Sensefinity, where REST is the standard for making requests. This layer is responsible for providing the basic endpoints for each smart contract specified previously to invoke the chaincode operations. By doing so, it simplifies the interaction with the blockchain network for end-users, hiding the underlying complexities of chaincode business logic.

7.2.1 Request Header

As discussed in previous chapters, each organization is responsible for inserting different assets into the blockchain. For each asset, a separate chaincode is employed to ensure separation of concerns, necessitating an API for each chaincode. To manage user identity and determine which peer to access, a request header is sent containing a JSON data structure with configuration variables for blockchain access (see Appendix A). These configuration variables include the identity of the person sending the transaction, comprising the user's public-private key pair and the MSP ID of the user's organization. Additionally, the peer client certificate is sent, as all nodes in the network operate with mutual TLS. Furthermore, information such as the peer endpoint, the channel, and the chaincode name is included, indicating the specific smart contract the user intends to interact with. Listing 7.3 provides an example of the request header.

```
{
  "msp_id": "CerfundaoPeerMSP",
  "client_cert": "/secrets/crt-admin0-cerfundao/versions/1",
  "client_key": "/secrets/key-admin0-cerfundao/versions/1",
  "ca_cert": "/secrets/ca-cerfundao/versions/1",
  "server_cert": "/secrets/crt-peer0-cerfundao/versions/1",
  "server_key": "/secrets/key-peer0-cerfundao/versions/1",
  "peer_endpoint": "peer0-cerfundao:8050",
  "gateway_peer": "peer0-cerfundao",
  "channel_name": "cherriesquality",
  "chaincode_name": "cherriesqualitycc"
}
```

Listing 7.3: Request Header

The rationale for using request headers lies in the need to incorporate configuration variables even within GET requests. GET requests typically reveal parameters in the URL as query parameters, which are visible and unsuitable for sensitive information. Furthermore, GET requests lack a request body where such data could be included, making request headers a practical alternative. Although the transmitted information isn't highly sensitive, it still encompasses crucial details such as the peer endpoint (a private IP), channel, and chaincode names. Additionally, certificates are referenced in their format, acting as pointers to certificates stored in Google Secret Manager alongside other certificates managed by Sensefinity. However, to ensure security, as it isn't safe to just send this information in an HTTP header [19], the request headers are secured by:

1. **Secure Communication:** Requests use HTTPS, encrypting data to prevent unauthorized interception or tampering.

2. **Token-Based Authentication:** Sensefinity API, which will call this blockchain API as explained in the next section, employs OAuth tokens to ensure only authorized users access the requested resources.

7.2.2 API Workflow

Figure 7.1 illustrates the communication messages exchanged between the participants during the process of creating an asset, specifically focusing on the addition of cherries by Cerfundão. The participants involved are Cerfundão, the REST API Layer, the Hyperledger Fabric Layer, and the Google Cloud Secret Manager. Each REST endpoint execution follows four steps:

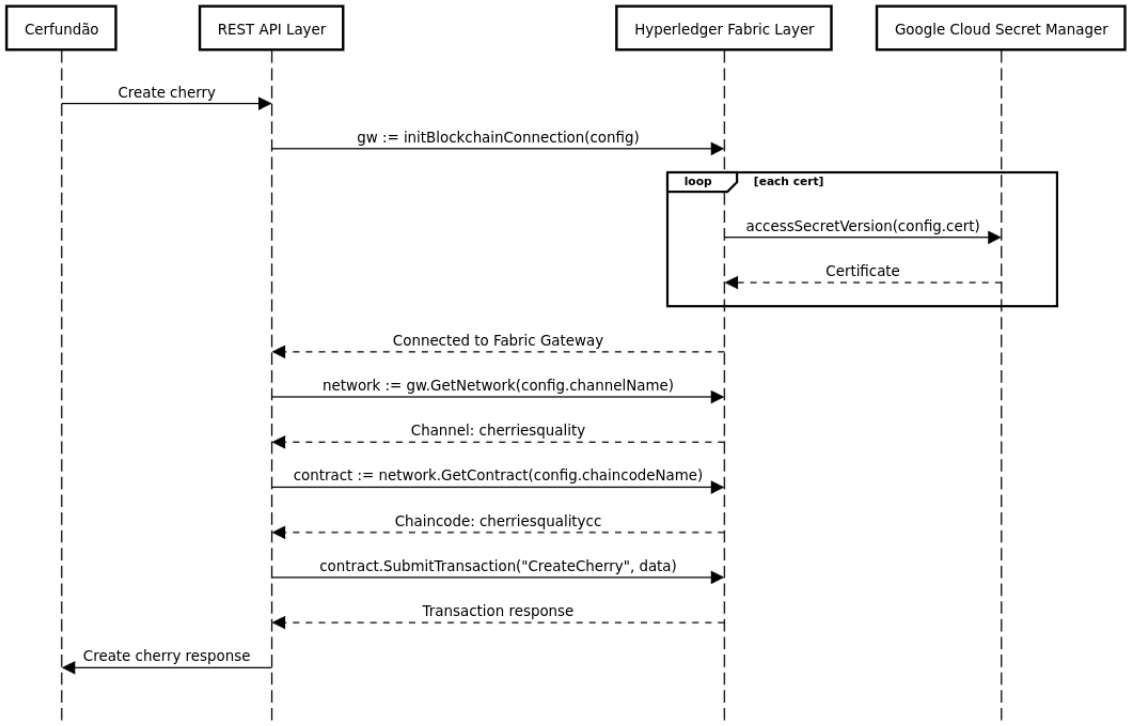

Figure 7.1: Creating an Asset: Sequence Diagram

1. **Establishing a connection:** The Fabric Gateway client API establishes a connection with the Fabric Gateway using the configuration passed in the request header. This is achieved through the **initBlockchainConnection(config)** function, which passes the request header formatted as a parameter. The Hyperledger Fabric Layer uses this configuration to create a gRPC connection to access the indicated peer. During this step, all necessary certificates are retrieved by interacting with the Google Cloud Secret Manager. This required a custom modification of the provided code, as the default implementation does not typically support fetching secrets from the Secret Manager.

2. **Accessing the network channel:** The next step involves accessing the network channel on which the operations need to be performed. This is done using the gateway connec-

tion established earlier and the **getNetwork(config.channelName)** function, which uses the channel name specified in the request header.

3. **Accessing the chaincode:** Subsequently, the specific chaincode is accessed using the network established in the previous step. This is done through the **getContract(config.chaincodeName)** function, which uses the chaincode name specified in the request header.

4. **Submitting the transaction:** If any of the previous steps fail due to network downtime, invalid credentials, or any other reason, the API will not submit a transaction to the network. However, if all steps are executed successfully, a transaction is prepared using the input arguments and submitted to the Fabric layer. This is done using the chaincode accessed in the previous step and the **submitTransaction("CreateCherry", data)** function, with the first parameter being the name of the smart contract to invoke in the chaincode and the second parameter being the data to be sent to the ledger. Once the chaincode is invoked and sends a response to the REST layer, the same response is relayed back to the user.

Executing a smart contract through the created REST API follows the same steps up to step 3. The difference in step 4 arises for read operations, where **evaluateTransaction("ReadCherryById", id)** is used instead of **submitTransaction()**. In this function, the first parameter is the name of the smart contract to be invoked, and the second parameter is the ID of the asset.

7.3 Creating the Application

The integration of blockchain functionalities into the Web Asset Management system, a GUI application for asset management developed by Sensefinity, began with the incorporation of new classes alongside existing ones from the Sensefinity domain model. Figure 7 highlights the newly added classes—Organization, Peer, Channel, Chaincode (all related to blockchain operations), and UserCertificates—in blue. Existing classes like User and Tenant, which are interconnected with these additions, are marked in grey. The attributes of these classes align with fields in the previously discussed request header and are implemented within the Sensefinity database using GORM, an ORM library for Golang [34]. Additionally, basic CRUD methods have been added to the existing Sensefinity REST API using Golang for each of these assets.

The introduction of these new classes into the existing system enables Sensefinity clients with an account in the Web Asset Management application to access blockchain-related asset information based on their permissions. Each user is associated with a Tenant, representing the client organizations within the company, similar to the concept of an Organization in the blockchain context. To ensure a clear separation between blockchain-related components and the existing system, an Organization class has been introduced. This class is linked with the Tenant class, ensuring independent management of blockchain-specific information while remaining integrated into the organizational structure. Given that blockchain access requires certificates, the UserCertificates class has been implemented to store the public-private key pairs for each user. The other

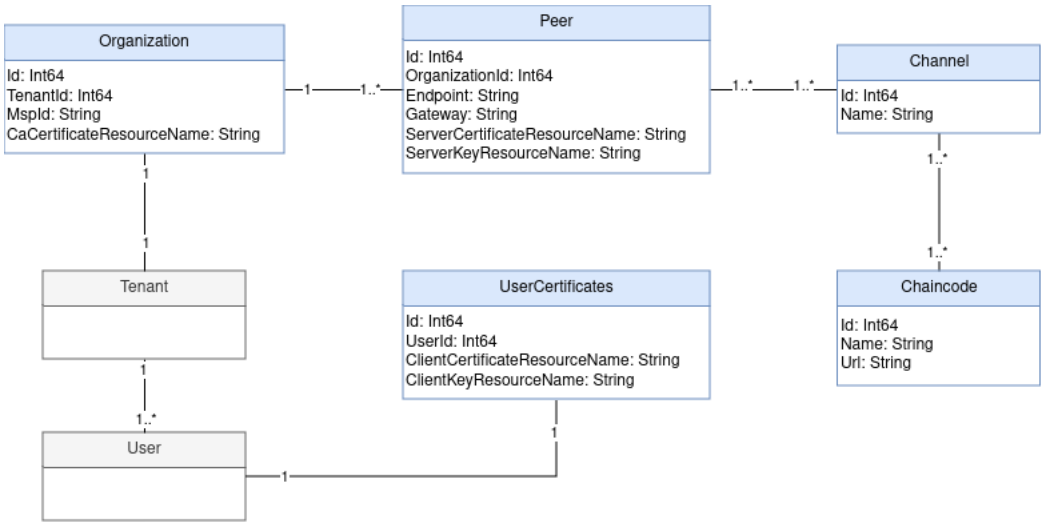

Figure 7.2: Application Domain Model

classes—Peer, Channel, and Chaincode—serve as containers for essential information needed to execute transactions on the blockchain. Access to the blockchain is facilitated through the Chaincode class, which includes a URL attribute specifying the API endpoint for interacting with the chaincode.

7.3.1 Application Overview

Figure 7.3 illustrates the page designed to list all transactions of an asset, displaying basic information such as asset ID, creator's certificate ID, and last modifier's certificate ID. This page was developed using ReactJS with the Metronic theme [36] already utilized by the company, and implemented in TypeScript.

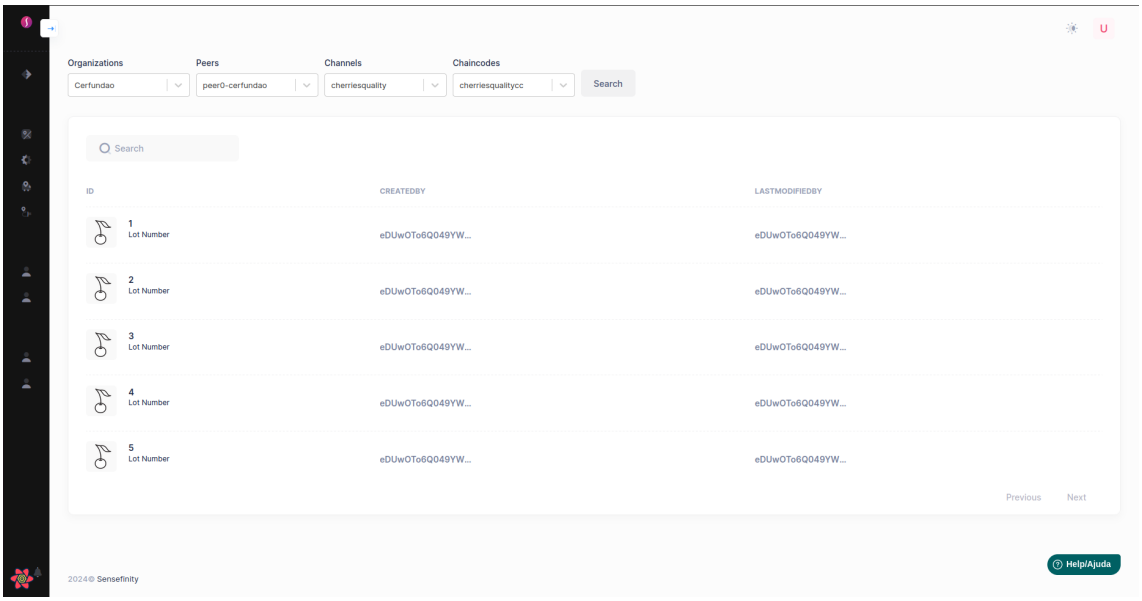

Figure 7.3: Web Asset Management: List of Transactions

The user is associated with their certificates through the UserCertificates class and belongs to the organization Cerfundão. Therefore, they possess certificates belonging to the MSP of this organization and have at least read operations permission (OU=user). Each time the user selects an option from a filter, such as choosing a peer, an API method executes to retrieve all peers of that organization stored in the database. This process is repeated for selecting channels and chaincodes. Once an option is chosen, another method fetches detailed information about that selection (e.g., the peer) and begins constructing the request header. Upon clicking the search button, the API utilizes the constructed request header along with the URL from the chaincode class to connect to the specific chaincode API and execute the GetAll smart contract for the cherry asset.

Figure 7.4 and Figure 7.5 illustrate the page where asset information recorded on the blockchain is displayed. The page comprises a header containing basic transaction information, including the transaction ID recorded on the blockchain. Additionally, it includes a body that presents two tab-panels:

The first tab-panel shows detailed asset information specific to the asset type. Since the request header was previously prepared on the listing page, a request is made to connect to the specific chaincode API. This request directly invokes the Read smart contract for this cherry asset using its ID and the pre-existing request header.

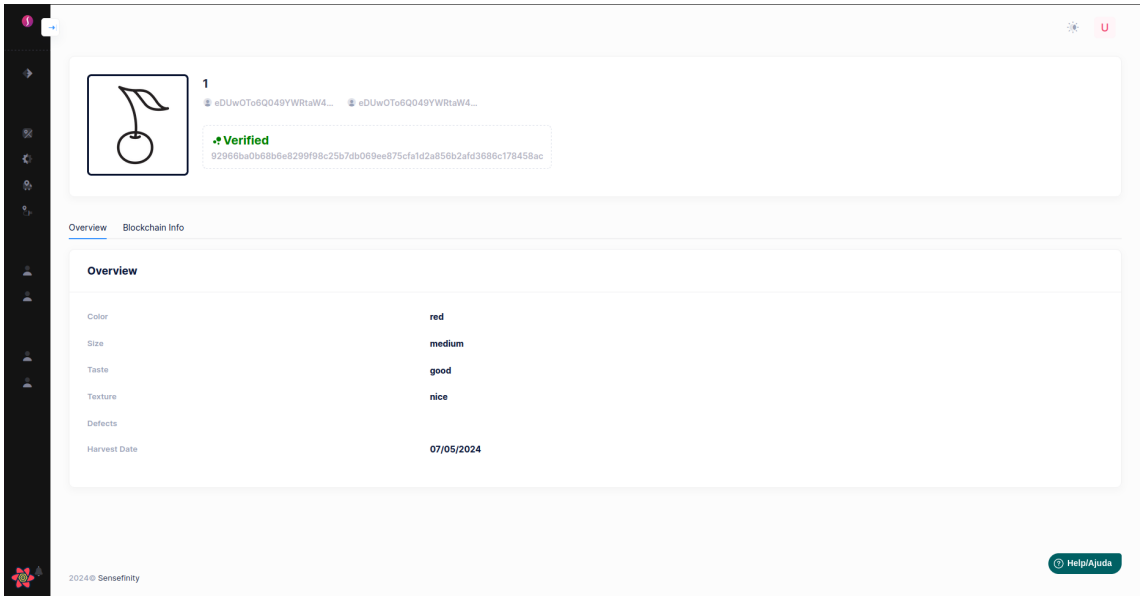

Figure 7.4: Web Asset Management: Asset Information

The second tab-panel displays blockchain-specific information about the transaction recorded on the blockchain. This includes the block number of the transaction, the current and previous hash blocks of the transaction, the transaction creator's MSP, and the MSPs of the organizations that endorsed the transaction before it was entered into the blockchain.

The chaincode name attribute from the Chaincode class is handled differently compared to other requests. Rather than retrieving it through calls to the Sensefinity API database, this specific blockchain information comes from the qscc system chaincode installed on every peer. Hence,

for these requests, the **chaincode_name** attribute in the request header is set to qscc. To obtain transaction-specific blockchain details, both the transaction ID (shown in the application header) and existing request header information are utilized. This process triggers the **GetTransaction-ByID** smart contract, exclusively available within the qscc chaincode, tailored specifically for retrieving direct blockchain transaction details.

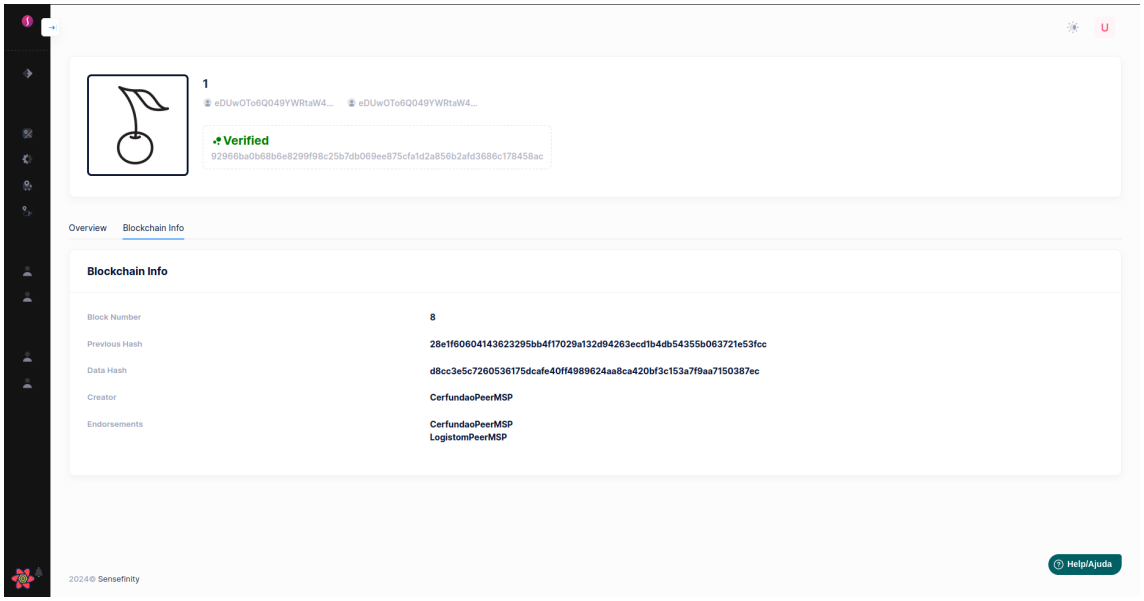

Figure 7.5: Web Asset Management: Transaction Information

7.4 Summary

In this chapter, the chaincode and its respective APIs were implemented to address functional requirements 1, 2, and 3: managing assets, retrieving asset history, and retrieving transactions. Although functional requirement 2 is not implemented, the approach for its implementation is explained.

For non-functional requirements related to security, both access control and secure communication are ensured. Access control is achieved by specifying which MSPs have permission to execute particular smart contracts, particularly in asset creation. Secure communication is upheld through HTTPS for the chaincode API and token-based authentication via the Sensefinity API.

Non-functional requirements related to performance are addressed through the deployment of both the chaincode and its respective API using Kubernetes, which ensures scalability and availability in line with the blockchain implementation. Deploying the chaincode as an external service satisfies the usability non-functional requirement for simplicity and supports modularity in maintainability. As with the blockchain implementation, documentation for the chaincode and API is available on Confluence.

The integration of blockchain functionalities into the Web Asset Management application by Sensefinity supports the goal of visualizing blockchain data in an application.

Chapter 8

Blockchain Benchmark

In this chapter, the objective is to test the performance of the implemented blockchain and understand the potential costs involved. The Hyperledger Caliper tool [22] is employed to conduct these tests. Caliper is a blockchain benchmarking tool that seamlessly integrates with Hyperledger Fabric, making it an ideal choice. The aim is to conduct three tests for reading and creating an asset in the blockchain, utilizing different types of machines, and covering various CPU and memory configurations. The chapter is divided into:

- **Section 8.1**: Explanation of the measurements used in the experiments.
- **Section 8.2**: Description of the test environment, including details on the machines used for the tests and where Caliper runs were conducted.
- **Section 8.3**: Discussion of the testing methodology, including the controller rates used for the experiments and the specific types of experiments conducted on each machine.
- **Section 8.4, 8.5, and 8.6**: Present the experiments conducted on each machine tested.
- **Section 8.7**: Provides a summary of the results of these experiments.

8.1 Performance Measurement

Hyperledger Caliper measures network performance using five key indicators:

- **Throughput (TPS):** The number of transactions submitted per second.
- **Send Rate (TPS):** The rate at which transactions are sent or submitted to the network.
- **Transaction Success and Failure Rates:** The rates at which transactions succeed or fail.
- **Latency (s):** The duration between issuing a transaction and receiving a response, indicating the time taken for transactions to be processed.
- **Average Resource Utilization:** The average level of resources (such as CPU and memory) used during the tests.

With these measures, the tests aim to identify scenarios where throughput aligns with the target TPS, indicating efficient resource utilization. If throughput exceeds the TPS, it suggests that the system is processing transactions efficiently, while lower throughput may indicate inefficiencies. Minor variations in throughput are expected, but significant deviations from the target TPS may signal issues with performance or test setup. Although Caliper supports resource utilization metrics, it is tailored for Docker containers. Since Kubernetes is being utilized, the metrics collector from Google Cloud is employed to capture CPU and memory metrics for the tests.

8.2 Test Environment

These tests are conducted in a staging environment within a Kubernetes cluster hosted on the Google Cloud platform. This staging environment is designed to closely mimic the production environment, ensuring that the benchmark tests executed closely resemble the reality in which the system will operate. The blockchain operates within a dedicated node pool, while Caliper is deployed in its dedicated node pool using machine type e2-standard-8. This separation prevents Caliper from competing for resources with the blockchain node pool, as confirmed by Caliper's resource usage metrics in Appendix E. For these tests, machines of the E2 series located in Madrid (europe-southwest on the Google Cloud) are used. The E2 series is known for its low cost of ownership and is primarily utilized for small to medium services. The tests are conducted on the following machines:

Machine Type	vCPUs	Memory (GB)	Estimated Price per Month ($)
e2-medium	2	4	120
e2-standard-4	4	16	190
e2-standard-8	8	32	305

Table 8.1: Machine Specifications and Pricing

All tests are executed using a single node, with auto-scaling disabled during these tests. For a production environment, deploying multiple nodes and enabling auto-scaling are recommended practices to enhance high availability.

8.3 Testing Methodology

The benchmark tests, conducted as part of this study, were identified by Al-Sumaidaee et al. [1] as essential for thoroughly assessing blockchain performance. These tests employ a fixed-rate controller to send input transactions at a constant rate, measured in transactions per second (TPS). The tests focus on evaluating the network under varying conditions by altering the number of workers (each representing a client connection), transaction volume, and transaction rate, as outlined below:

1. **Assessment of Worker Variability:** This test evaluates how the blockchain network's per-

formance varies with different numbers of workers, each representing a client connection submitting transactions or queries. Increasing the number of workers simulates higher user demand and network participation, providing insight into the network's scalability under increased load. The results help identify the optimal number of workers for each machine type by analyzing performance metrics.

2. **Transaction Volume Analysis:** This test examines the system's performance over varying total transaction volumes to assess its ability to maintain consistent performance over time. By observing the system's behavior under sustained load, the test identifies potential bottlenecks and evaluates long-term scalability and resource utilization.

3. **Transaction Rate Impact Assessment:** This test examines the impact of varying transaction rates on the performance of the blockchain network. It seeks to identify the transaction rate threshold at which failures occur, which is essential for determining the network's capacity limits. The test ultimately establishes the maximum transactions per second (TPS) that the network can effectively manage.

All payloads used in the tests were randomly generated, ranging from 1 KB to 8 KB, to realistically simulate the variability of actual data. Transactions were sent to the channel where cherry-related information is stored, using the smart contracts **CreateCherry** and **ReadCherry** for creating and reading data, respectively. This channel requires endorsement from two endorsers.

8.4 Machine 1 (e2-medium)

8.4.1 Test 1 (Workers Number)

In this experiment, presented in Tables 8.2 and 8.3, the number of workers is varied between 5, 15, and 25, while keeping the transaction number constant at 1000 for CreateCherry, aiming for a transaction rate of 10 transactions per second (TPS). Similarly, for ReadCherry, the transaction number is adjusted to 10,000, targeting a consistent TPS of 100. The results demonstrate:

1. **For CreateCherry:**

   - Transaction throughput and send rate remain relatively stable at around 10 transactions per second.
   - Latency slightly increases from 0.44 to 0.64 seconds.
   - CPU usage and memory usage both show a slight increase, with CPU usage rising from 0.53 to 0.63 and memory usage increasing from 1.26 to 1.39.

2. **For ReadCherry:**

   - Transaction throughput and send rate remain high and consistent around 100 transactions per second.

- Latency remains very low, with minor fluctuations from 0.01 to 0.02 seconds.
- CPU and memory usage increase, from 0.56 to 0.70 for CPU and from 1.24 to 1.36 for memory, indicating higher processing demand similar to CreateCherry.

In conclusion, the experiments revealed that increasing the number of workers from 5 to 25 resulted in only minor performance variations for both CreateCherry and ReadCherry. The 5-worker configuration was the most efficient, with lower resource consumption. Higher worker counts were associated with increased latency. Therefore, the 5-worker setup is recommended for subsequent tests to ensure optimal results and resource utilization.

CreateCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
5	1000	10	0.44	10.0	10.0	0.53	1.26
15	1000	10	0.56	10.1	10.1	0.56	1.26
25	1000	10	0.64	10.1	10.2	0.63	1.39

Table 8.2: CreateCherry - Workers Number (e2-medium)

ReadCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
5	10000	100	0.01	100.0	100.0	0.56	1.24
15	10000	100	0.02	99.8	99.8	0.63	1.39
25	10000	100	0.02	99.4	99.4	0.70	1.36

Table 8.3: ReadCherry - Workers Number (e2-medium)

8.4.2 Test 2 (Transactions Number)

In this experiment, presented in Tables 8.4 and 8.5, the number of workers remains constant at 5, while the transaction number for CreateCherry is varied between 100, 250, and 1000, all targeting a transaction rate of 10 transactions per second (TPS). Similarly, for ReadCherry, the transaction number is kept at 10,000, ensuring a consistent TPS of 100. The results reveal:

1. **For CreateCherry:**
   - Transaction throughput and send rate remain relatively stable at around 10 transactions per second.
   - Latency varies slightly, ranging from 0.49 to 0.45 seconds, with minor fluctuations, showing a decrease as the transaction number increases.
   - CPU usage and memory usage both experience slight increases, from 0.32 to 0.49 for CPU and from 1.19 to 1.33 for memory.

2. **For ReadCherry:**

   - Transaction throughput and send rate remain high and consistent at around 100 transactions per second.
   - Latency remains consistently low, fluctuating minimally from 0.01 to 0.02 seconds.
   - CPU usage slightly increases from 0.44 to 0.49, similar to CreateCherry, and memory usage also sees a slight rise, from 1.26 to 1.36.

In conclusion, the tests revealed an unexpected decrease in latency as the number of transactions increased, potentially due to the concurrency and parallelism mechanisms within Hyperledger Fabric. Further investigation is needed to understand this behavior and optimize system performance accordingly.

CreateCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
5	100	10	0.49	10.3	10.5	0.32	1.19
5	250	10	0.46	10.1	10.2	0.44	1.26
5	1000	10	0.45	10.0	10.1	0.49	1.33

Table 8.4: CreateCherry - Transactions Number (e2-medium)

ReadCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
5	1000	100	0.01	99.9	99.9	0.16	1.20
5	10000	100	0.01	100.0	100.0	0.51	1.26
5	25000	100	0.01	100.0	100.0	0.62	1.35

Table 8.5: ReadCherry - Transactions Number (e2-medium)

8.4.3 Test 3 (TPS Rate)

In this experiment, presented in Tables 8.6 and 8.7, the number of workers is maintained at 5. For CreateCherry, the transaction number remains constant at 1000, corresponding to TPS rates of 10, 15, and 20, respectively. Similarly, ReadCherry sustains a transaction number of 10,000 across different TPS rates: 100, 150, and 200. The findings are summarized as follows:

1. **For CreateCherry:**

   - Transaction throughput and send rate remain relatively stable in the tests that were conducted with 10 and 15 TPS.
   - Latency decreases as the TPS rate increases, aligning with the observed trend in the transaction number test, indicating improved efficiency with higher transaction volumes.

- At 20 TPS, a slight decrease in throughput is observed, coinciding with an increase in the failure rate.

2. **For ReadCherry:**

- Transaction throughput and send rate remain stably consistent across all tests.
- Latency remains consistently low, exhibiting minor fluctuations as the TPS rate varies.

At higher TPS rates, like 20 TPS for CreateCherry, there's a decrease in throughput and an increase in the failure rate. This suggests that the system starts to struggle to handle the increased load efficiently beyond certain thresholds. Such observations underscore the importance of identifying optimal TPS rates to maintain system stability and performance. This is attributed to the absence of an endorsement plan, which overwhelmed endorsing peers and caused delays or failures in meeting endorsement policy requirements.

CreateCherry								
N. of Workers	**Tx Number**	**Fail**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
5	1000	0	10	0.48	10.0	10.0	0.35	1.31
5	1000	0	15	0.39	15.0	15.1	0.46	1.40
5	1000	2	20	0.37	19.3	20.1	0.52	1.29

Table 8.6: CreateCherry - TPS Rate (e2-medium)

ReadCherry								
N. of Workers	**Tx Number**	**Fail**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
5	10000	0	100	0.01	100.0	100.0	0.48	1.35
5	10000	0	150	0.01	150.0	150.0	0.58	1.42
5	10000	0	200	0.02	199.6	199.7	0.72	1.28

Table 8.7: ReadCherry - TPS Rate (e2-medium)

8.5 Machine 2 (e2-standard-4)

8.5.1 Test 1 (Workers Number)

In this experiment, outlined in Tables 8.8 and 8.9, the number of workers varied across configurations of 25, 35, and 50. For CreateCherry, the transaction number was fixed at 2000, targeting a consistent transaction rate of 25 transactions per second (TPS). Similarly, ReadCherry maintained a transaction number of 20,000 across different worker configurations, aiming for a stable TPS of 200.

1. **For CreateCherry:**

- Transaction throughput and send rate demonstrate stability, maintaining around 25 transactions per second across different worker configurations.

2. **For ReadCherry:**

- Transaction throughput and send rate remain consistently high at approximately 200 transactions per second across varying worker counts of 25 and 35.
- At 50 workers, a slight decrease in throughput is observed while the send rate remains consistent, indicating potential system limitations.

Latency and resource consumption mirror the initial machine type. At 50 workers, there is a slight decrease in throughput, suggesting a potential bottleneck or limitation in the system's capacity to handle the increased workload. However, the consistent send rate indicates that the system is still able to process transactions at the same rate they are being sent, despite the decreased throughput.

The 25-worker configuration is the most stable, chosen for subsequent tests.

CreateCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
25	2000	25	0.35	25.0	25.0	0.66	1.34
35	2000	25	0.36	24.7	25.1	0.67	1.39
50	2000	25	0.42	24.9	24.9	0.69	1.45

Table 8.8: CreateCherry - Workers Number (e2-standard-4)

ReadCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
25	20000	200	0.01	200.0	200.0	0.76	1.27
35	20000	200	0.01	198.6	198.6	0.78	1.51
50	20000	200	0.01	196.4	196.4	0.89	1.53

Table 8.9: ReadCherry - Workers Number (e2-standard-4)

8.5.2 Test 2 (Transactions Number)

In this experiment, depicted in Tables 8.10 and 8.11, the number of workers remained constant at 25 across all configurations. For CreateCherry, the transaction numbers were set at 500, 3000, and 6000, aiming for a stable transaction rate of 25 transactions per second (TPS). Similarly, ReadCherry sustained transaction numbers of 30,000, 40,000, and 100,000, respectively, targeting a consistent TPS of 300.

1. **For CreateCherry:**

- Transaction throughput and send rate remain consistently high at approximately 25 transactions per second across all transaction number variations.

2. **For ReadCherry:**

- Transaction throughput and send rate remain consistently high at around 300 transactions per second across the variations in transaction numbers, specifically at 30,000 and 40,000 transactions.
- At the 100,000 transaction variation, a slight decrease in throughput is observed, suggesting potential system limitations.

In this test, it becomes more apparent that lower transaction numbers lead to higher throughput and send rates compared to higher transactions. The system manages many transactions without failures, but processing time increases with transaction numbers.

CreateCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
25	500	25	0.36	25.4	25.5	0.33	1.63
25	3000	25	0.31	25.0	25.1	0.90	1.68
25	6000	25	0.28	25.0	25.0	0.95	1.87

Table 8.10: CreateCherry - Transactions Number (e2-standard-4)

ReadCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
25	30000	300	0.01	299.0	299.1	1.19	1.86
25	40000	300	0.01	299.4	299.4	1.05	1.72
25	100000	300	0.01	296.0	296.2	1.04	1.79

Table 8.11: ReadCherry - Transactions Number (e2-standard-4)

8.5.3 Test 3 (TPS Rate)

In this experiment, depicted in Tables 8.12 and 8.13, the number of workers remained constant at 25 for all test configurations. For CreateCherry, the transaction number was consistently set at 2000, while targeting transaction rates of 25, 30, and 35 TPS. For ReadCherry, transaction numbers were maintained at 20,000, aiming for TPS targets of 300, 400, and 500.

1. **For CreateCherry:**

- Transaction throughput and send rate remain relatively stable in the tests conducted at 25, and 30 TPS.

- At 35 TPS, a slight increase in failure rate is observed, coinciding with a slight increase in throughput.

2. **For ReadCherry:**

   - Transaction throughput and send rate remain consistently high at 300 transactions per second, but at 400 and 500 TPS, there is a significant decrease in throughput, suggesting potential system limitations.

The results obtained demonstrate a limit of 30 TPS for CreateCherry without experiencing transaction failures. However, for ReadCherry, as the TPS increases, the throughput does not fully align with the TPS, although the system handles a significant volume of reads, which is noteworthy, particularly at 500 TPS.

CreateCherry								
N. of Workers	**Tx Number**	**Fail**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
25	2000	0	25	0.33	25.0	25.1	0.66	1.34
25	2000	0	30	0.29	30.0	30.0	0.91	1.85
25	2000	5	35	0.29	32.5	35.0	0.75	1.89

Table 8.12: CreateCherry - TPS Rate (e2-standard-4)

ReadCherry								
N. of Workers	**Tx Number**	**Fail**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
25	20000	0	300	0.01	298.9	298.9	1.19	1.86
25	20000	0	400	0.03	396.4	396.5	1.13	1.82
25	20000	0	500	0.08	493.9	494.0	1.26	1.94

Table 8.13: ReadCherry - TPS Rate (e2-standard-4)

8.6 Machine 3 (e2-standard-8)

8.6.1 Test 1 (Workers Number)

In this experiment, highlighted in Tables 8.14 and 8.15, the number of workers was adjusted to 50, 75, and 85 configurations. CreateCherry was tested with a fixed transaction number of 3000, aiming for a stable transaction rate of 30 transactions per second (TPS). Similarly, ReadCherry maintained a consistent transaction number of 30,000, targeting a TPS of 300 across different worker counts.

1. **For CreateCherry:**

   - Transaction throughput and send rate remain consistently stable at approximately 30 transactions per second across configurations with 50 and 75 workers.

- At 85 workers, there is a noticeable decrease in Transaction Throughput (TPS), indicating potential system limitations.

2. **For ReadCherry:**

   - TPS and send rate remain high at 300 transactions per second for 50 and 75 workers. However, the 85-worker configuration shows a noticeable decrease in TPS, suggesting system limitations.

The 75-worker configuration is the most stable and chosen for subsequent tests.

CreateCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
50	3000	30	0.19	30.3	30.3	1.12	1.29
75	3000	30	0.28	30.0	30.0	1.16	1.48
85	3000	30	0.29	29.5	30.0	1.39	1.17

Table 8.14: CreateCherry - Workers Number (e2-standard-8)

ReadCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
50	30000	300	0.01	299.1	299.1	1.58	1.51
75	30000	300	0.01	299.0	299.0	1.68	1.57
85	30000	300	0.01	297.8	297.8	1.75	1.58

Table 8.15: ReadCherry - Workers Number (e2-standard-8)

8.6.2 Test 2 (Transactions Number)

In this evaluation, as documented in Tables 8.10 and 8.11, the number of workers remained constant at 75 across all tests. CreateCherry processed fixed transaction numbers of 3000, 6000, and 10000, targeting a consistent transaction rate of 40 transactions per second (TPS). Similarly, ReadCherry handled transaction counts of 30,000, 60,000, and 100,000, aiming for a consistent TPS of 400.

1. **For CreateCherry:**

   - Transaction throughput remains consistent at around 40 transactions per second across various transaction numbers (3,000, 6,000, and 10,000).

2. **For ReadCherry:**

   - Transaction throughput and send rate remain consistently high at around 400 transactions per second across the 30,000 and 60,000 transaction volumes.

- At the 100,000 transaction volume, a noticeable decrease in throughput to 395.6 TPS is observed, indicating potential system limitations under higher loads.

In this test, a similar pattern observed on the previous e2-standard-4 machine emerges, where system throughput begins to decrease at 100,000 transaction volumes. This suggests that the limitation might originate from the Hyperledger Fabric itself. Despite the high transaction volumes, TPS, and number of workers employed, the system's CPU and memory usage remain relatively modest compared to the machine's specifications.

CreateCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
75	3000	40	0.25	39.9	39.9	1.49	1.37
75	6000	40	0.23	40.0	40.1	2.24	1.83
75	10000	40	0.23	39.8	40.1	2.24	2.00

Table 8.16: CreateCherry - Transactions Number (e2-standard-8)

ReadCherry							
N. of Workers	**Tx Number**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
75	30000	400	0.01	399.3	399.3	1.96	1.62
75	60000	400	0.01	398.8	398.8	2.12	1.72
75	100000	400	0.01	395.6	395.6	2.73	2.05

Table 8.17: ReadCherry - Transactions Number (e2-standard-8)

8.6.3 Test 3 (TPS Rate)

In this experiment, depicted in Tables 8.12 and 8.13, the number of workers remained constant at 75 for all test configurations. For CreateCherry, the transaction number was consistently set at 3000, while targeting transaction rates of 45, 50, and 60 TPS. For ReadCherry, transaction numbers were maintained at 30,000, aiming for TPS targets of 120, 135, and 150.

1. **For CreateCherry:**
   - Transaction throughput and send rate remain relatively stable in the tests conducted at 45 and 50 TPS.
   - At 60 TPS, a slight decrease in throughput is observed, coinciding with a slight increase in failure rate.
2. **For ReadCherry:**
   - Transaction throughput and send rate remain consistently high at 500 and 600 TPS, but at 700 TPS, a notable decrease in throughput is observed, suggesting potential system limitations.

The results indicate a limitation of 50 TPS for CreateCherry, without experiencing transaction failures being the constraint. Conversely, for ReadCherry, the throughput remains extremely high, similar to other tests, but starts to decrease at 700 TPS.

CreateCherry								
N. of Workers	**Tx Number**	**Fail**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
75	3000	0	45	0.24	44.8	44.8	1.45	1.22
75	3000	0	50	0.24	49.9	49.9	1.65	1.35
75	3000	1	60	0.23	57.6	59.4	1.85	1.50

Table 8.18: CreateCherry - TPS Rate (e2-standard-8)

ReadCherry								
N. of Workers	**Tx Number**	**Fail**	**TPS**	**Latency**	**Throughput**	**Send Rate**	**CPU**	**Memory**
75	30000	0	500	0.01	499.5	499.5	1.80	1.55
75	30000	0	600	0.01	598.7	598.6	1.90	1.70
75	30000	0	700	0.01	695.2	695.1	1.95	1.75

Table 8.19: ReadCherry - TPS Rate (e2-standard-8)

8.7 Summary

The results of these tests satisfy the performance non-functional requirement of efficiency, successfully recording over 60 transactions per minute with low latency across all machines tested. For Machine 1 (e2-medium), the best performance was achieved with 5 workers, handling up to 15 TPS before transaction failures began. Machine 2 (e2-standard-4) performed optimally with 25 workers, supporting up to 30 TPS, while Machine 3 (e2-standard-8) showed the best results with 75 workers, managing up to 50 TPS without significant issues, demonstrating the scalability of each machine.

Increasing the number of workers led to higher latency, while higher transaction volumes unexpectedly resulted in reduced latency, likely due to the concurrency and parallelism mechanisms within Hyperledger Fabric, which warrant further investigation. However, as TPS increased, transaction failures became more frequent, confirmed to be due to delays in the endorsement process. Read operations remained lightweight with consistently low latency, even under high volumes. Nonetheless, performance degradation was observed as transaction volumes approached 100,000, indicating a potential bottleneck within Hyperledger Fabric. With these insights, the Sensefinity team can now develop a comprehensive system cost plan for their customers, with the estimated monthly price calculated using the Google Cloud Calculator [18].

Chapter 9

Conclusion

The first phase of this project was successfully completed, encompassing the design and implementation of a traceability system for the Cerejas do Fundão supply chain. This phase involved setting up the blockchain infrastructure for each organization within the consortium and deploying the system using Kubernetes on Google Cloud to enhance availability and scalability. Key components of this phase included the development of an API for interacting with blockchain-managed smart contracts, addressing functional requirements such as asset management and transaction retrieval. An application was also created to visualize cherry-related data sourced from the blockchain. Additionally, robust access control mechanisms were established to ensure data confidentiality and security within the blockchain environment. Comprehensive documentation and monitoring tools, including Prometheus and Grafana, were integrated to track system metrics and maintain operational oversight. Detailed information on the deployment process using Kubernetes and Google Cloud is available in the Appendix C to keep the main report concise.

Benchmark tests were conducted, yielding favorable results. The testing phase assessed the blockchain architecture's performance under various configurations and transaction rates. The architecture demonstrated strong efficiency and scalability: Machine 1 (e2-medium) efficiently handled up to 15 transactions per second (TPS) with 5 workers before experiencing failures; Machine 2 (e2-standard-4) supported up to 30 TPS with 25 workers; and Machine 3 (e2-standard-8) managed up to 50 TPS with 75 workers. While increasing the number of workers led to higher latency, higher transaction volumes surprisingly reduced latency, likely due to effective concurrency mechanisms. Despite the generally low latency for read operations, performance began to degrade as transaction volumes approached 100,000, revealing a potential bottleneck. The project has now advanced to real-world testing, allowing for refined validation of these findings and the potential adjustment of the architecture to address any issues identified, particularly given the lower-cost configuration used to control expenses.

9.0.1 Future Work

Although the first phase achieved several key milestones, certain aspects remain incomplete. The non-functional requirement for interoperability has yet to be met, as the system is not yet integrated with the consortium's existing systems. Currently, data is manually entered into the blockchain by

each organization within the consortium, with plans for automation in the next phase. Additionally, sensor data will be incorporated throughout the traceability process to enhance accuracy and transparency. It is crucial to understand the specific data that each consortium member intends to send to the blockchain and to determine what information can be shared with other stakeholders. This is particularly relevant for Sonae, which gathered the least amount of information during this phase. This also raises the question of whether the use of channels will be sufficient to protect sensitive information from other organizations or if more complex measures, such as private data collections, will be required to safeguard information shared within a channel by some participants.

Furthermore, while the functional requirements for managing assets and transaction retrieval have been implemented, the requirement for retrieving asset history remains to be completed. Although the approach for this implementation is explained in the report, it has not yet been executed.

The need to achieve full decentralization, as required by the project, will be further explored. The implementation of IPFS for decentralized storage will be considered to address the risks associated with the current centralized storage system using NFS. Additionally, a more in-depth study of the concurrency and parallelism mechanisms in Hyperledger Fabric is necessary to better understand some of the benchmark results, particularly those related to transaction volume.

Acronyms

DLT Distributed Ledger Technology. 13, 14

EVM Ethereum Virtual Machine. 16

gRPC Google Remote Procedure Call. 9, 38, 62

IPFS InterPlanetary File System. 35, 36, 80

MSP Membership Service Provider. 10, 32, 35, 42, 44, 50, 59, 65

MVCC Multi-Version Concurrency Control. 11

NFS Network File System. 35, 36, 42, 50, 80

PBFT Practical Byzantine Fault Tolerance. 8, 9, 16–19, 22

PoA Proof of Authority. 16

PoS Proof of Stake. 8

PoW Proof of Work. 8, 16

PV Persistent Volume. 13, 42

PVC Persistent Volume Claim. 13, 42

Bibliography

[1] Ghassan Al-Sumaidaee, Rami Alkhudary, Zeljko Zilic, and Andraws Swidan. Performance analysis of a private blockchain network built on hyperledger fabric for healthcare. *Information Processing & Management*, 60(2):103160, 2023.

[2] Thomas Allweyer. *BPMN 2.0: introduction to the standard for business process modeling*. BoD–Books on Demand, 2016.

[3] AMA – Agência para a Modernização Administrativa. Portugal's Recovery and Resilience Plan. https://transparencia.gov.pt/pt/fundos-europeus/prr/beneficiarios-projetos/projeto/02/C05-i01.01/2022.PC644918095-00000033/. Accessed: 2024-06-27.

[4] 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.

[5] Atlassian. Confluence. https://www.atlassian.com/software/confluence. Accessed: 2024-08-19.

[6] Juan Benet. Ipfs-content addressed, versioned, p2p file system. *arXiv preprint arXiv:1407.3561*, 2014.

[7] Alison Blay-Palmer. *Food fears: From industrial to sustainable food systems*. Routledge, 2016.

[8] Sotirios Brotsis, Nicholas Kolokotronis, Konstantinos Limniotis, Gueltoum Bendiab, and Stavros Shiaeles. On the security and privacy of hyperledger fabric: Challenges and open issues. In *2020 IEEE World Congress on Services (SERVICES)*, pages 197–204. IEEE, 2020.

[9] Richard Gendal Brown, James Carlyle, Ian Grigg, and Mike Hearn. Corda: an introduction. *R3 CEV, August*, 1(15):14, 2016.

[10] Vitalik Buterin et al. Ethereum white paper. *GitHub repository*, 1:22–23, 2013.

[11] Miguel Castro, Barbara Liskov, et al. Practical byzantine fault tolerance. In *OsDI*, volume 99, pages 173–186, 1999.

[12] Jeeta Ann Chacko, Ruben Mayer, and Hans-Arno Jacobsen. Why do my blockchain transactions fail? a study of hyperledger fabric. In *Proceedings of the 2021 international conference on management of data*, pages 221–234, 2021.

[13] Ian Curry. Version 3 x. 509 certificates. *Entrust Technologies White Paper*, 1996.

[14] Thomas K Dasaklis, Theodore G Voutsinas, Giannis T Tsoulfas, and Fran Casino. A systematic literature review of blockchain-enabled supply chain traceability implementations. *Sustainability*, 14(4):2439, 2022.

[15] John Estdale and Elli Georgiadou. Applying the iso/iec 25010 quality models to software product. In *Systems, Software and Services Process Improvement: 25th European Conference, EuroSPI 2018, Bilbao, Spain, September 5-7, 2018, Proceedings 25*, pages 492–503. Springer, 2018.

[16] David Freet, Rajeev Agrawal, Sherin John, and Jessie J Walker. Cloud forensics challenges from a service model standpoint: Iaas, paas and saas. In *Proceedings of the 7th International Conference on Management of computational and collective intElligence in Digital EcoSystems*, pages 148–155, 2015.

[17] Gartner. 20% top WW grocers to use blockchain for food safety by 2025. https://www.gartner.com/en/newsroom/press-releases/2019-04-30-gartner-predicts-20-percent-of-top-global-grocers-wil. Accessed: 2023-09-10.

[18] Google Cloud. Google Cloud Pricing Calculator. https://cloud.google.com/products/calculator. Accessed: 2024-03-16.

[19] David Gourley and Brian Totty. *HTTP: the definitive guide*. " O'Reilly Media, Inc.", 2002.

[20] GS1 in Europe. Digital product passport – GS1 in europe. https://gs1.eu/activities/digital-product-passport/. Accessed: 2023-09-10.

[21] Christine V Helliar, Louise Crawford, Laura Rocca, Claudio Teodori, and Monica Veneziani. Permissionless and permissioned blockchain diffusion. *International Journal of Information Management*, 54:102136, 2020.

[22] Hyperledger. Hyperledger Caliper. https://www.hyperledger.org/projects/caliper. Accessed: 2024-03-16.

[23] Hyperledger. Hyperledger Fabric CA Client Configuration. https://hyperledger-fabric-ca.readthedocs.io/en/release-1.4/clientconfig.html. Accessed: 2024-03-16.

[24] Hyperledger. Hyperledger Fabric CA Server Configuration. https://hyperledger-fabric-ca.readthedocs.io/en/release-1.4/serverconfig.html. Accessed: 2024-03-16.

[25] Hyperledger. Hyperledger Fabric Documentation: Creating Channel Configuration. https://hyperledger-fabric.readthedocs.io/en/release-2.5/create_channel/create_channel_config.html. Accessed: 2024-03-16.

[26] Hyperledger. Hyperledger Fabric Gateway. https://hyperledger.github.io/fabric-gateway/. Accessed: 2024-07-17.

[27] Hyperledger. Orderer Deployment - Hyperledger Fabric. https://hyperledger-fabric.readthedocs.io/en/release-2.5/deployorderer/ordererdeploy.html. Accessed: 2024-03-16.

[28] Hyperledger. Peer Deployment - Hyperledger Fabric. https://hyperledger-fabric.readthedocs.io/en/release-2.5/deploypeer/peerdeploy.html. Accessed: 2024-03-16.

[29] Hyperledger Fabric. Hyperledger Fabric 2.5 Ordering Service Documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.5/orderer/ordering_service.html. Accessed: 2024-07-04.

[30] Hyperledger Fabric. Hyperledger Fabric Deployment Guide. https://hyperledger-fabric.readthedocs.io/en/latest/deployment_guide_overview.html. Accessed: 2024-07-18.

[31] Hyperledger Fabric. Hyperledger Fabric SDK Chaincode Documentation. https://hyperledger-fabric.readthedocs.io/en/release-2.5/sdk_chaincode.html. Accessed: 2024-08-21.

[32] Instituto Nacional de Estatística. Recenseamento agrícola. análise dos principais resultados: 2019. https://www.ine.pt/xurl/pub/437178558. Accessed: 2023-12-16.

[33] S. Jain. *Programming Hyperledger Fabric: Creating Enterprise Blockchain Applications*. SIDDHARTH JAIN, 2020.

[34] Jinzhu Zhang. GORM - The fantastic ORM library for Golang. https://gorm.io/index.html. Accessed: 2024-06-12.

[35] Reshma Kamath. Food traceability on blockchain: Walmart's pork and mango pilots with ibm. *The Journal of the British Blockchain Association*, 1(1), 2018.

[36] KeenThemes. Metronic 8 - Quick Start Guide for React. https://preview.keenthemes.com/metronic8/react/docs/quick-start. Accessed: 2024-06-18.

[37] Kubernetes. Gateway API. https://gateway-api.sigs.k8s.io/. Accessed: 2024-07-18.

[38] Kubernetes. Kubernetes. https://kubernetes.io/. Accessed: 2024-05-18.

[39] Ivan Laishevskiy, Artem Barger, and Vladimir Gorgadze. A journey towards the most efficient state database for hyperledger fabric. In *2023 IEEE International Conference on Blockchain and Cryptocurrency (ICBC)*, pages 1–3. IEEE, 2023.

[40] Jun Lin, Zhiqi Shen, Anting Zhang, and Yueting Chai. Blockchain and iot based food traceability for smart agriculture. In *Proceedings of the 3rd international conference on crowd science and engineering*, pages 1–6, 2018.

[41] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. *Decentralized business review*, 2008.

[42] Nestlé. Nestlé Implements Blockchain Technology for Zoégas Coffee Brand. https://www.nestle.com/media/news/nestle-blockchain-zoegas-coffee-brand. Accessed: 2024-06-27.

[43] Brian Pawlowski, Chet Juszczak, Peter Staubach, Carl Smith, Diane Lebel, and Dave Hitz. Nfs version 3: Design and implementation. In *USENIX Summer*, pages 137–152. Boston, MA, 1994.

[44] Julien Polge, Jérémy Robert, and Yves Le Traon. Permissioned blockchain frameworks in the industry: A comparison. *Ict Express*, 7(2):229–233, 2021.

[45] Dewmini Premarathna. Reduce food crop wastage with hyperledger fabric-based food supply chain. In *2021 International Research Conference on Smart Computing and Systems Engineering (SCSE)*, volume 4, pages 168–176. IEEE, 2021.

[46] Sara Saberi, Mahtab Kouhizadeh, Joseph Sarkis, and Lejia Shen. Blockchain technology and its relationships to sustainable supply chain management. *International journal of production research*, 57(7):2117–2135, 2019.

[47] Lakshmi Siva Sankar, M Sindhu, and M Sethumadhavan. Survey of consensus protocols on blockchain applications. In *2017 4th international conference on advanced computing and communication systems (ICACCS)*, pages 1–5. IEEE, 2017.

[48] Sensefinity. Sensefinity. https://www.sensefinity.com/. Accessed: 2024-06-28.

[49] Marios Touloupou, Marinos Themistocleous, Elias Iosif, and Klitos Christodoulou. A systematic literature review toward a blockchain benchmarking framework. *IEEE Access*, 10:70630–70644, 2022.

[50] Achilleas Tzenetopoulos, Dimosthenis Masouros, Nikolaos Kapsoulis, Antonios Litke, Dimitrios Soudris, and Theodora Varvarigou. Hlf-kubed: Blockchain-based resource monitoring for edge clusters. *Ledger*, 7, 2022.

[51] Mueen Uddin. Blockchain medledger: Hyperledger fabric enabled drug traceability system for counterfeit drugs in pharmaceutical industry. *International Journal of Pharmaceutics*, 597:120235, 2021.

[52] Shan Wang, Ming Yang, Yue Zhang, Yan Luo, Tingjian Ge, Xinwen Fu, and Wei Zhao. On private data collection of hyperledger fabric. In *2021 IEEE 41st International Conference on Distributed Computing Systems (ICDCS)*, pages 819–829. IEEE, 2021.

[53] Karl Wüst and Arthur Gervais. Do you need a blockchain? In *2018 crypto valley conference on blockchain technology (CVCBT)*, pages 45–54. IEEE, 2018.

[54] Gyeongsik Yang, Kwanhoon Lee, Kyungwoon Lee, Yeonho Yoo, Hyowon Lee, and Chuck Yoo. Resource analysis of blockchain consensus algorithms in hyperledger fabric. *IEEE Access*, 10:74902–74920, 2022.

[55] Xinting Yang, Mengqi Li, Huajing Yu, Mingting Wang, Daming Xu, and Chuanheng Sun. A trusted blockchain-based traceability system for fruit and vegetable agricultural products. *IEEE Access*, 9:36282–36293, 2021.

[56] Zibin Zheng, Shaoan Xie, Hongning Dai, Xiangping Chen, and Huaimin Wang. An overview of blockchain technology: Architecture, consensus, and future trends. In *2017 IEEE international congress on big data (BigData congress)*, pages 557–564. Ieee, 2017.

Appendix A

Data Structures

A.1 Cherries

```
type Cherries struct {
    CherryID string           `json:"cherry_id"`
    OutgoingBatchID string    `json:"outgoing_batch_id"`
    IncomingBatchID string    `json:"incoming_batch_id"`
    DateTime time.Time        `json:"date_time"`
    Size string               `json:"size"`
    Color string              `json:"color"`
    Weight float64            `json:"weight"`
    Producer string           `json:"producer"`
    Plot string               `json:"plot"`
}
```

Listing A.1: Data Structure - Cherries

A.2 Logistics

```
type Logistics struct {
    LogisticID string         `json:"logistic_id"`
    TransportGuideID string   `json:"transport_guide_id"`
    DateTime time.Time        `json:"date_time"`
    Transporter string        `json:"transporter"`
    VehicleID string          `json:"vehicle_id"`
    Route string              `json:"route"`
    EstimatedArrival time.Time `json:"estimated_arrival"`
}
```

Listing A.2: Data Structure - Logistics

A.3 Payments

```
type Payments struct {
    PaymentID string          `json:"payment_id"`
    Amount float64            `json:"amount"`
    Currency string           `json:"currency"`
    PaymentDate time.Time     `json:"payment_date"`
    Status string             `json:"status"`
    Method string             `json:"method"`
    Payer string              `json:"payer"`
```

```
    Receiver string           `json:"receiver"`
}
```

Listing A.3: Data Structure - Payments

A.4 BlockchainConfig

```
type BlockchainConfig struct {
  MspID         string `json:"msp_id"`
  ClientCert    string `json:"client_cert"`
  ClientKey     string `json:"client_key"`
  CaCert        string `json:"ca_cert"`
  ServerCert    string `json:"server_cert"`
  ServerKey     string `json:"server_key"`
  PeerEndpoint  string `json:"peer_endpoint"`
  GatewayPeer   string `json:"gateway_peer"`
  ChannelName   string `json:"channel_name"`
  ChaincodeName string `json:"chaincode_name"`
}
```

Listing A.4: Data Structure - BlockchainConfig

Appendix B

Configuration Files

B.1 CA Server Configuration

```
version: 1.5.7

port: 6000

debug: false

crlsizelimit: 512000

tls:
  enabled: true
  certfile:
  keyfile:
  clientauth:
    type: noclientcert
    certfiles:

ca:
  name: "Sensefinity Orderer CA"
  keyfile:
  certfile:
  chainfile:

crl:
  expiry: 24h

registry:
  maxenrollments: -1
  identities:
     - name: sensefinityorderer
       pass: sensefinityordererpw
       type: client
       affiliation: ""
       attrs:
          hf.Registrar.Roles: "*"
          hf.Registrar.DelegateRoles: "*"
          hf.Revoker: true
          hf.IntermediateCA: true
          hf.GenCRL: true
          hf.Registrar.Attributes: "*"
          hf.AffiliationMgr: true

db:
```

```
  type: postgres
  datasource: host=postgres-ca port=5432 user=postgres password=postgres dbname
    =ca sslmode=disable
  tls:
      enabled: false
      certfiles:
      client:
        certfile:
        keyfile:

ldap:
   enabled: false
   url: ldap://<adminDN>:<adminPassword>@<host>:<port>/<base>
   tls:
      certfiles:
      client:
         certfile:
         keyfile:
   attribute:
      names: ['uid','member']
      converters:
         - name:
           value:
      maps:
         groups:
            - name:
              value:

affiliations:

signing:
    default:
      usage:
        - digital signature
      expiry: 8760h
    profiles:
      ca:
         usage:
           - cert sign
           - crl sign
         expiry: 43800h
         caconstraint:
           isca: true
           maxpathlen: 0
      tls:
         usage:
            - signing
            - key encipherment
            - server auth
            - client auth
            - key agreement
         expiry: 8760h

csr:
   cn: "Sensefinity Orderer CA"
   names:
      - C: PT
        ST: "Campo Grande"
        L: "Lisboa"
        O: "Sensefinity Orderer"
        OU:
```

```
   hosts:
     - localhost
     - ca-orderermsp-sensefinity
   ca:
      expiry: 131400h
      pathlength: 1

bccsp:
    default: SW
    sw:
        hash: SHA2
        security: 256
        filekeystore:
            keystore: msp/keystore

cacount:

cafiles:

intermediate:
  parentserver:
    url:
    caname:
  enrollment:
    hosts:
    profile:
    label:
  tls:
    certfiles:
    client:
      certfile:
      keyfile:
```

Listing B.1: CA Server Config

B.2 Orderer Configuration

```
General:
    ListenAddress: 0.0.0.0
    ListenPort: 7050
    TLS:
        Enabled: true
        PrivateKey: tls/server.key
        Certificate: tls/server.crt
        RootCAs:
          - tls/ca.crt
        ClientAuthRequired: true
        ClientRootCAs:
          - tls/ca.crt
    Keepalive:
        ServerMinInterval: 60s
        ServerInterval: 7200s
        ServerTimeout: 20s
    Cluster:
        SendBufferSize: 10
        ClientCertificate: tls/server.crt
        ClientPrivateKey: tls/server.key
        ListenPort: 7070
        ListenAddress: 0.0.0.0
```

```
        ServerCertificate: tls/server.crt
        ServerPrivateKey: tls/server.key
        RootCAs: tls/ca.crt
    BootstrapMethod: file
    BootstrapFile: genesis.block
    LocalMSPDir: msp
    LocalMSPID: SensefinityOrdererMSP
    Profile:
        Enabled: false
        Address: 0.0.0.0:6060
    BCCSP:
        Default: SW
        SW:
            Hash: SHA2
            Security: 256
            FileKeyStore:
                KeyStore:
        PKCS11:
            Library:
            Label:
            Pin:
            Hash:
            Security:
            FileKeyStore:
                KeyStore:
    Authentication:
        TimeWindow: 15m

FileLedger:
    Location: /var/hyperledger/production
    Prefix: hyperledger-fabric-ordererledger

Kafka:
    Retry:
        ShortInterval: 5s
        ShortTotal: 10m
        LongInterval: 5m
        LongTotal: 12h
        NetworkTimeouts:
            DialTimeout: 10s
            ReadTimeout: 10s
            WriteTimeout: 10s
        Metadata:
            RetryBackoff: 250ms
            RetryMax: 3
        Producer:
            RetryBackoff: 100ms
            RetryMax: 3
        Consumer:
            RetryBackoff: 2s
    Topic:
        ReplicationFactor: 3
    Verbose: false
    TLS:
        Enabled: false
        PrivateKey:
        Certificate:
        RootCAs:
    SASLPlain:
      Enabled: false
      User:
```

```
      Password:
    Version:

Debug:
    BroadcastTraceDir:
    DeliverTraceDir:

Operations:
    ListenAddress: 0.0.0.0:8443
    TLS:
        Enabled: false
        Certificate:
        PrivateKey:
        ClientAuthRequired: false
        ClientRootCAs: []

Metrics:
    Provider: prometheus
    Statsd:
      Network: udp
      Address: 127.0.0.1:8125
      WriteInterval: 30s
      Prefix:

Consensus:
    WALDir: /var/hyperledger/production/etcdraft/wal
    SnapDir: /var/hyperledger/production/etcdraft/snapshot
```

Listing B.2: Orderer Config

B.3 Peer Configuration

```
peer:
    id: peer0-cerfundao
    networkId: prod
    listenAddress: 0.0.0.0:8050
    chaincodeListenAddress: 0.0.0.0:8051
    chaincodeAddress: peer0-sonae:8051
    address: peer0-sonae:8050
    addressAutoDetect: false
    keepalive:
        interval: 7200s
        timeout: 20s
        minInterval: 60s
        client:
            interval: 60s
            timeout: 20s
        deliveryClient:
            interval: 60s
            timeout: 20s
    gossip:
        bootstrap: peer0-cerfundao:8050
        endpoint: peer0-cerfundao:8050
        externalEndpoint: peer0-cerfundao.cerfundao.svc.cluster.local:8050
        useLeaderElection: true
        orgLeader: false
        membershipTrackerInterval: 5s
        maxBlockCountToStore: 100
        maxPropagationBurstLatency: 10ms
```

```
        maxPropagationBurstSize: 10
        propagateIterations: 1
        propagatePeerNum: 3
        pullInterval: 4s
        pullPeerNum: 3
        requestStateInfoInterval: 4s
        publishStateInfoInterval: 4s
        stateInfoRetentionInterval:
        publishCertPeriod: 10s
        skipBlockVerification: false
        dialTimeout: 3s
        connTimeout: 2s
        recvBuffSize: 20
        sendBuffSize: 200
        digestWaitTime: 1s
        requestWaitTime: 1500ms
        responseWaitTime: 2s
        aliveTimeInterval: 5s
        aliveExpirationTimeout: 25s
        reconnectInterval: 25s
        election:
            startupGracePeriod: 15s
            membershipSampleInterval: 1s
            leaderAliveThreshold: 10s
            leaderElectionDuration: 5s
        pvtData:
            pullRetryThreshold: 60s
            transientstoreMaxBlockRetention: 1000
            pushAckTimeout: 3s
            btlPullMargin: 10
            reconcileBatchSize: 10
            reconcileSleepInterval: 1m
            reconciliationEnabled: true
            skipPullingInvalidTransactionsDuringCommit: false
        state:
            enabled: true
            checkInterval: 10s
            responseTimeout: 3s
            batchSize: 10
            blockBufferSize: 100
            maxRetries: 20
    tls:
        enabled: true
        clientAuthRequired: true
        cert:
            file: tls/server.crt
        key:
            file: tls/server.key
        rootcert:
            file: tls/ca.crt
        clientRootCAs:
            files:
              - tls/ca.crt
        clientKey:
            file: tls/server.key
        clientCert:
            file: tls/server.crt
    authentication:
        timewindow: 15m
    fileSystemPath: /var/hyperledger/production
    BCCSP:
```

```
        Default: SW
        SW:
            Hash: SHA2
            Security: 256
            FileKeyStore:
                KeyStore:
        PKCS11:
            Library:
            Label:
            Pin:
            Hash:
            Security:
    mspConfigPath: msp
    localMspId: CerfundaoPeerMSP
    client:
        connTimeout: 3s
    deliveryclient:
        reconnectTotalTimeThreshold: 3600s
        connTimeout: 3s
        reConnectBackoffThreshold: 3600s
        addressOverrides:
    localMspType: bccsp
    profile:
        enabled: false
        listenAddress: 0.0.0.0:6060
    handlers:
        authFilters:
          -
            name: DefaultAuth
          -
            name: ExpirationCheck
        decorators:
          -
            name: DefaultDecorator
        endorsers:
          escc:
            name: DefaultEndorsement
            library:
        validators:
          vscc:
            name: DefaultValidation
            library:
    validatorPoolSize:
    discovery:
        enabled: true
        authCacheEnabled: true
        authCacheMaxSize: 1000
        authCachePurgeRetentionRatio: 0.75
        orgMembersAllowedAccess: false
    limits:
      concurrency:
        qscc: 5000

vm:
    endpoint: unix:///host/var/run/docker.sock
    docker:
        tls:
            enabled: false
            ca:
                file: docker/ca.crt
            cert:
```

```
                file: docker/tls.crt
            key:
                file: docker/tls.key
        attachStdout: false
        hostConfig:
            NetworkMode: host
            Dns:
            LogConfig:
                Type: json-file
                Config:
                    max-size: "50m"
                    max-file: "5"
            Memory: 2147483648

chaincode:
    id:
        path:
        name:
    builder: $(DOCKER_NS)/fabric-ccenv:$(TWO_DIGIT_VERSION)
    pull: false
    golang:
        runtime: $(DOCKER_NS)/fabric-baseos:$(TWO_DIGIT_VERSION)
        dynamicLink: false
    java:
        runtime: $(DOCKER_NS)/fabric-javaenv:$(TWO_DIGIT_VERSION)
    node:
        runtime: $(DOCKER_NS)/fabric-nodeenv:$(TWO_DIGIT_VERSION)
    externalBuilders:
     - name: external-builder
       path: /home/external-builder
    installTimeout: 300s
    startuptimeout: 300s
    executetimeout: 30s
    mode: net
    keepalive: 0
    system:
        _lifecycle: enable
        cscc: enable
        lscc: enable
        escc: enable
        vscc: enable
        qscc: enable
    logging:
      level:  info
      shim:   warning
      format: '%{color}%{time:2006-01-02 15:04:05.000 MST} [%{module}] %{
    shortfunc} -> %{level:.4s} %{id:03x}%{color:reset} %{message}'

ledger:
  blockchain:
  state:
    stateDatabase: CouchDB
    totalQueryLimit: 100000
    couchDBConfig:
       couchDBAddress: localhost:5984
       username: user
       password: password
       maxRetries: 20
       maxRetriesOnStartup: 12
       requestTimeout: 35s
       internalQueryLimit: 1000
```

```
      maxBatchUpdateSize: 1000
      warmIndexesAfterNBlocks: 1
      createGlobalChangesDB: false
      cacheSize: 64
  history:
    enableHistoryDatabase: true
  pvtdataStore:
    collElgProcMaxDbBatchSize: 5000
    collElgProcDbBatchesInterval: 1000

operations:
    listenAddress: 0.0.0.0:9443
    tls:
        enabled: false
        cert:
            file:
        key:
            file:
        clientAuthRequired: false
        clientRootCAs:
            files: []

metrics:
    provider: prometheus
    statsd:
        network: udp
        address: 127.0.0.1:8125
        writeInterval: 10s
        prefix:
```

Listing B.3: Peer Config

B.4 Peer CLI Configuration

```
peer:
    address: peer0-cerfundao:8050
    mspConfigPath: msp
    localMspId: CerfundaoPeerMSP
    localMspType: bccsp
    tls:
        enabled: true
        clientAuthRequired: true
        rootcert:
            file: tls/ca.crt
        clientRootCAs:
            files:
              - tls/ca.crt
        clientKey:
            file: tls/server.key
        clientCert:
            file: tls/server.crt
    keepalive:
        interval: 7200s
        timeout: 20s
        minInterval: 60s
    client:
        interval: 60s
        timeout: 20s
    BCCSP:
```

```
        Default: SW
        SW:
            Hash: SHA2
            Security: 256
            FileKeyStore:
                KeyStore:
        PKCS11:
            Library:
            Label:
            Pin:
            Hash:
            Security:

ledger:
  blockchain:
  state:
    stateDatabase: CouchDB
    totalQueryLimit: 100000
    couchDBConfig:
       couchDBAddress: localhost:5984
       username: user
       password: password
       maxRetries: 20
       maxRetriesOnStartup: 12
       requestTimeout: 35s
       internalQueryLimit: 1000
       maxBatchUpdateSize: 1000
       warmIndexesAfterNBlocks: 1
       createGlobalChangesDB: false
       cacheSize: 64
  history:
    enableHistoryDatabase: true
  pvtdataStore:
    collElgProcMaxDbBatchSize: 5000
    collElgProcDbBatchesInterval: 1000

operations:
    listenAddress: 0.0.0.0:9443
    tls:
        enabled: false
        cert:
            file:
        key:
            file:
        clientAuthRequired: false
        clientRootCAs:
            files: []

metrics:
    provider: prometheus
    statsd:
        network: udp
        address: 127.0.0.1:8125
        writeInterval: 10s
        prefix:
```

Listing B.4: Peer CLI Config

Appendix C

Deployment Files

C.1 CA Server Deployment

```
FROM hyperledger/fabric-ca:1.5

WORKDIR /home

ENV FABRIC_CA_HOME=$WORKDIR

COPY ./docker.yaml ./fabric-ca-server-config.yaml
COPY ./tls-cert.pem ./tls-cert.pem

EXPOSE 6000

CMD ["fabric-ca-server", "start"]
```

Listing C.1: CA Server Dockerfile

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: ca-orderermsp-sensefinity
  namespace: sensefinity
spec:
  selector:
    matchLabels:
      app: ca-orderermsp-sensefinity
  replicas: 1
  template:
    metadata:
      labels:
        app: ca-orderermsp-sensefinity
    spec:
      affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                    - key: cloud.google.com/gke-nodepool
                      operator: In
                      values:
                        - "sensefinity-dev-blockchain-node-pool"
      containers:
        - name: ca-orderermsp-sensefinity
          image: image_to_be_replaced
```

```
          imagePullPolicy: "Always"
```

Listing C.2: CA Server Deploy

```
apiVersion: v1
kind: Service
metadata:
  name: ca-orderermsp-sensefinity
  namespace: sensefinity
  labels:
    app: ca-orderermsp-sensefinity
spec:
  type: ClusterIP
  selector:
    app: ca-orderermsp-sensefinity
  ports:
    - protocol: TCP
      targetPort: 6000
      port: 6000
```

Listing C.3: CA Server Service

C.2 Orderer Deployment

```
FROM hyperledger/fabric-orderer:2.5.0

ENV FABRIC_CFG_PATH=/home
ENV ORDERER_FILELEDGER_LOCATION=/var/hyperledger/production
ENV ORDERER_CONSENSUS_WALDIR=/var/hyperledger/production/etcdraft/wal
ENV ORDERER_CONSENSUS_SNAPDIR=/var/hyperledger/production/etcdraft/snapshot

WORKDIR /home

COPY ./prod-genesis.block ./genesis.block
COPY ./msp ./msp
COPY ./tls ./tls
COPY ./orderer.yaml .

EXPOSE 7050 7070 8443

CMD ["orderer"]
```

Listing C.4: Orderer Dockerfile

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: orderer0-sensefinity
  namespace: sensefinity
spec:
  selector:
    matchLabels:
      name: orderer0-sensefinity
  replicas: 1
  template:
    metadata:
      labels:
        name: orderer0-sensefinity
    spec:
```

```
      affinity:
        nodeAffinity:
          requiredDuringSchedulingIgnoredDuringExecution:
            nodeSelectorTerms:
              - matchExpressions:
                  - key: cloud.google.com/gke-nodepool
                    operator: In
                    values:
                      - "sensefinity-dev-blockchain-node-pool"
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: pvc-sensefinity
      containers:
        - image: image_to_be_replaced
          name: orderer0-sensefinity
          imagePullPolicy: Always
          volumeMounts:
            - name: data
              mountPath: /var/hyperledger/production
              subPath: data/orderers/orderer0
```

Listing C.5: Orderer Deploy

```
kind: Service
apiVersion: v1
metadata:
  name: orderer0-sensefinity
  namespace: sensefinity
  labels:
    run: orderer0-sensefinity
spec:
  selector:
    name: orderer0-sensefinity
  type: ClusterIP
  ports:
    - protocol: TCP
      port: 7050
      name: grpc
    - protocol: TCP
      port: 7070
      name: raft
---
#---------------- Orderer0 Metrics Service ---------------#
apiVersion: v1
kind: Service
metadata:
  labels:
    app: orderer0-sensefinity
    metrics-service: "true"
  name: metrics-orderer0-sensefinity
  namespace: sensefinity
spec:
  type: ClusterIP
  ports:
  - name: "orderer-metrics"
    port: 8443
    targetPort: 8443
  selector:
    name: orderer0-sensefinity
```

Listing C.6: Orderer Service

C.3 Peer Deployment

```
FROM hyperledger/fabric-peer:2.5.0

ENV FABRIC_CFG_PATH=/home
ENV FABRIC_LOGGING_SPEC=info
ENV GODEBUG=netdns=go

WORKDIR /home

RUN apt-get update && apt-get install -y jq

EXPOSE 8050 9443

COPY ./msp ./msp
COPY ./tls ./tls
COPY ./docker.yaml ./core.yaml
COPY ./external-builder ./external-builder

CMD ["peer", "node", "start"]
```

Listing C.7: Peer Dockerfile

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: peer0-cerfundao
  namespace: cerfundao
spec:
  selector:
    matchLabels:
      name: peer0-cerfundao
  replicas: 1
  template:
    metadata:
      labels:
        name: peer0-cerfundao
    spec:
      affinity:
          nodeAffinity:
            requiredDuringSchedulingIgnoredDuringExecution:
              nodeSelectorTerms:
                - matchExpressions:
                    - key: cloud.google.com/gke-nodepool
                      operator: In
                      values:
                        - "sensefinity-dev-blockchain-node-pool"
      volumes:
        - name: data
          persistentVolumeClaim:
            claimName: pvc-cerfundao
      containers:
        - name: peer0-cerfundao
          image: image_to_be_replaced
          volumeMounts:
            - name: data
              mountPath: /var/hyperledger/production
              subPath: data/peers/peer0
        - name: couchdb-peer0-cerfundao
          image: couchdb:3.3.3
          volumeMounts:
```

```
            - name: data
              mountPath: /opt/couchdb/data
              subPath: data/couchdb/couchdb-peer0
          env:
          - name: COUCHDB_USER
            value: user
          - name: COUCHDB_PASSWORD
            value: password
```

Listing C.8: Peer Deploy

```
kind: Service
apiVersion: v1
metadata:
  name: peer0-cerfundao
  namespace: cerfundao
  labels:
    app: peer0-cerfundao
spec:
  selector:
    name: peer0-cerfundao
  type: ClusterIP
  ports:
    - name: grpc
      port: 8050
      protocol: TCP
    - name: couchdb
      port: 5984
      protocol: TCP
---
#---------------- Peer0 Metrics Service ---------------#
apiVersion: v1
kind: Service
metadata:
  labels:
    app: peer0-cerfundao
    metrics-service: "true"
  name: metrics-peer0-cerfundao
  namespace: cerfundao
spec:
  type: ClusterIP
  ports:
  - name: "peer-metrics"
    port: 9443
    targetPort: 9443
  selector:
    name: peer0-cerfundao
```

Listing C.9: Peer Service

C.4 Peer CLI Deployment

```
FROM hyperledger/fabric-tools:2.5

ENV FABRIC_CFG_PATH=/home
ENV FABRIC_LOGGING_SPEC=info
ENV GODEBUG=netdns=go

WORKDIR /home
```

```
8
9  COPY ./msp ./msp
10 COPY ./tls ./tls
11 COPY ./docker.yaml ./core.yaml
12
13 CMD ["/bin/bash"]
```

Listing C.10: Peer CLI Dockerfile

```
1  apiVersion: apps/v1
2  kind: Deployment
3  metadata:
4    name: cli-peer0-cerfundao
5    namespace: cerfundao
6  spec:
7    selector:
8      matchLabels:
9        name: cli-peer0-cerfundao
10   template:
11     metadata:
12       labels:
13         name: cli-peer0-cerfundao
14     spec:
15       affinity:
16           nodeAffinity:
17             requiredDuringSchedulingIgnoredDuringExecution:
18               nodeSelectorTerms:
19                 - matchExpressions:
20                     - key: cloud.google.com/gke-nodepool
21                       operator: In
22                       values:
23                         - "sensefinity-dev-blockchain-node-pool"
24       volumes:
25         - name: data
26           persistentVolumeClaim:
27             claimName: pvc-cerfundao
28       containers:
29         - name: cli-peer0-cerfundao
30           stdin: true
31           tty: true
32           image: image_to_be_replaced
33           volumeMounts:
34             - name: data
35               mountPath: /home/aux
36               subPath: aux
```

Listing C.11: Peer CLI Deploy

C.5 Auto Scaling

```
1  apiVersion: autoscaling.k8s.io/v1
2  kind: VerticalPodAutoscaler
3  metadata:
4    name: vpa-peer0-cerfundao
5    namespace: cerfundao
6  spec:
7    targetRef:
8      apiVersion: apps/v1
9      kind: Deployment
```

```
    name: peer0-cerfundao
  updatePolicy:
    updateMode: "Auto"
```

Listing C.12: Vertical Pod Autoscaler (VPA)

C.6 Service Exposure

```
apiVersion: gateway.networking.k8s.io/v1beta1
kind: HTTPRoute
metadata:
  name: http-cerfundao
  namespace: cerfundao
spec:
  parentRefs:
    - name: gw-blockchain
      namespace: infra
      kind: Gateway
  rules:
    - matches:
      - path:
          type: PathPrefix
          value: /cerfundao/ca/peermsp
      filters:
        - type: URLRewrite
          urlRewrite:
            path:
              type: ReplacePrefixMatch
              replacePrefixMatch: /
      backendRefs:
      - name: ca-peermsp-cerfundao
        port: 6003
    - matches:
      - path:
          type: PathPrefix
          value: /cerfundao/ca/orderermsp
      filters:
        - type: URLRewrite
          urlRewrite:
            path:
              type: ReplacePrefixMatch
              replacePrefixMatch: /
      backendRefs:
      - name: ca-orderermsp-cerfundao
        port: 6002
    - matches:
        - path:
            type: PathPrefix
            value: /cerfundao/api
      filters:
        - type: URLRewrite
          urlRewrite:
            path:
              type: ReplacePrefixMatch
              replacePrefixMatch: /
      backendRefs:
        - name: api-cerfundao
          port: 3000
```

Listing C.13: HTTPRoute

Appendix D

Smart Contracts

D.1 Create

```
func (s *SmartContract) CreateCherry(ctx contractapi.
    TransactionContextInterface, cherryRaw string) error {
  cherry := types.CherryQuality{}
  err := json.Unmarshal([]byte(cherryRaw), &cherry)
  if err != nil {
    return err
  }

  mspID, err := ctx.GetClientIdentity().GetMSPID()
  if err != nil {
    return err
  }
  if mspID != "CerfundaoPeerMSP" {
    return fmt.Errorf("insufficient permissions")
  }

  txId := ctx.GetStub().GetTxID()
  cherry.TransactionID = txId

  clientID, err := ctx.GetClientIdentity().GetID()
  if err != nil {
    return err
  }

  exists, err := s.CherryExists(ctx, cherry.ID)
  if err != nil {
    return err
  }
  if exists {
    return fmt.Errorf("the cherry %s already exists", cherry.ID)
  }

  cherry.CreatedBy = clientID
  cherry.LastModifiedBy = clientID

  cherryJSON, err := json.Marshal(cherry)
  if err != nil {
    return err
  }

  return ctx.GetStub().PutState(cherry.ID, cherryJSON)
```

```
}
```

Listing D.1: Smart Contract - Create

D.2 Update

```
func (s *SmartContract) UpdateCherry(ctx contractapi.
    TransactionContextInterface, cherryRaw string) error {
  cherry := types.CherryQuality{}
  err := json.Unmarshal([]byte(cherryRaw), &cherry)
  if err != nil {
    return err
  }

  mspID, err := ctx.GetClientIdentity().GetMSPID()
  if err != nil {
    return err
  }
  if mspID != "CerfundaoPeerMSP" {
    return fmt.Errorf("insufficient permissions")
  }

  clientID, err := ctx.GetClientIdentity().GetID()
  if err != nil {
    return err
  }

  exists, err := s.CherryExists(ctx, cherry.ID)
  if err != nil {
    return err
  }
  if !exists {
    return fmt.Errorf("the cherry %s does not exist", cherry.ID)
  }

  cherry.LastModifiedBy = clientID

  cherryJSON, err := json.Marshal(cherry)
  if err != nil {
    return err
  }

  return ctx.GetStub().PutState(cherry.ID, cherryJSON)
}
```

Listing D.2: Smart Contract - Update

D.3 Read

```
func (s *SmartContract) ReadCherry(ctx contractapi.TransactionContextInterface,
    id string) (*types.CherryQuality, error) {
  cherryJSON, err := ctx.GetStub().GetState(id)
  if err != nil {
    return nil, fmt.Errorf("failed to read from world state: %v", err)
  }
  if cherryJSON == nil {
    return nil, fmt.Errorf("the cherry %s does not exist", id)
```

```
  }

  var cherry types.CherryQuality
  err = json.Unmarshal(cherryJSON, &cherry)
  if err != nil {
    return nil, err
  }

  return &cherry, nil
}
```

Listing D.3: Smart Contract - Read

D.4 Read All

```
func (s *SmartContract) GetAllCherries(ctx contractapi.
    TransactionContextInterface) ([]*types.CherryQuality, error) {
  resultsIterator, err := ctx.GetStub().GetStateByRange("", "")
  if err != nil {
    return nil, err
  }
  defer resultsIterator.Close()

  var cherries []*types.CherryQuality
  for resultsIterator.HasNext() {
    queryResponse, err := resultsIterator.Next()
    if err != nil {
      return nil, err
    }

    var cherry types.CherryQuality
    err = json.Unmarshal(queryResponse.Value, &cherry)
    if err != nil {
      return nil, err
    }
    cherries = append(cherries, &cherry)
  }

  return cherries, nil
}
```

Listing D.4: Smart Contract - Read All

D.5 Delete

```
func (s *SmartContract) DeleteCherry(ctx contractapi.
    TransactionContextInterface, id string) error {
  mspID, err := ctx.GetClientIdentity().GetMSPID()
  if err != nil {
    return err
  }
  if mspID != "CerfundaoPeerMSP" {
    return fmt.Errorf("insufficient permissions")
  }

  exists, err := s.CherryExists(ctx, id)
  if err != nil {
```

```
    return err
  }
  if !exists {
    return fmt.Errorf("the cherry %s does not exist", id)
  }

  return ctx.GetStub().DelState(id)
}
```

Listing D.5: Smart Contract - Delete

D.6 Exists

```
func (s *SmartContract) CherryExists(ctx contractapi.
    TransactionContextInterface, id string) (bool, error) {
  cherryJSON, err := ctx.GetStub().GetState(id)
  if err != nil {
    return false, fmt.Errorf("failed to read from world state: %v", err)
  }

  return cherryJSON != nil, nil
}
```

Listing D.6: Smart Contract - Exists

Appendix E

Caliper Resource Usage

E.1 Machine 1 (e2-medium)

E.1.1 Test 1 (Workers Number)

CreateCherry		
N. of Workers	**Max. CPU**	**Max. Memory**
5	0.38	0.63
15	0.81	1.33
25	1.55	1.40

Table E.1: CreateCherry - Workers Number (e2-medium) for Caliper

ReadCherry		
N. of Workers	**Max. CPU**	**Max. Memory**
5	0.48	0.60
15	0.68	1.16
25	0.77	1.58

Table E.2: ReadCherry - Workers Number (e2-medium) for Caliper

E.1.2 Test 2 (Transactions Number)

CreateCherry		
Tx Number	**Max. CPU**	**Max. Memory**
100	0.28	0.59
250	0.46	0.66
1000	0.41	0.64

Table E.3: CreateCherry - Transactions Number (e2-medium) for Caliper

ReadCherry		
Tx Number	**Max. CPU**	**Max. Memory**
1000	0.20	0.51
10000	0.52	0.65
25000	0.58	0.61

Table E.4: ReadCherry - Transactions Number (e2-medium) for Caliper

E.1.3 Test 3 (TPS Rate)

CreateCherry		
TPS	**Max. CPU**	**Max. Memory**
10	0.23	0.48
15	0.40	0.62
20	0.57	0.64

Table E.5: CreateCherry - TPS Rate (e2-medium) for Caliper

ReadCherry		
TPS	**Max. CPU**	**Max. Memory**
100	0.61	0.59
150	0.60	0.61
200	0.63	0.48

Table E.6: ReadCherry - TPS Rate (e2-medium) for Caliper

E.2 Machine 2 (e2-standard-4)

E.2.1 Test 1 (Workers Number)

CreateCherry		
N. of Workers	**Max. CPU**	**Max. Memory**
25	1.43	2.12
35	2.21	2.85
50	2.48	3.92

Table E.7: CreateCherry - Workers Number (e2-standard-4) for Caliper

ReadCherry		
N. of Workers	**Max. CPU**	**Max. Memory**
25	0.95	1.80
35	1.49	2.38
50	1.65	3.38

Table E.8: ReadCherry - Workers Number (e2-standard-4) for Caliper

E.2.2 Test 2 (Transactions Number)

CreateCherry		
Tx Number	**Max. CPU**	**Max. Memory**
500	1.26	1.83
3000	1.36	2.19
6000	1.47	2.14

Table E.9: CreateCherry - Transactions Number (e2-standard-4) for Caliper

ReadCherry		
Tx Number	**Max. CPU**	**Max. Memory**
30000	1.50	2.30
40000	1.13	2.22
100000	1.79	2.11

Table E.10: ReadCherry - Transactions Number (e2-standard-4) for Caliper

E.2.3 Test 3 (TPS Rate)

CreateCherry		
TPS	**Max. CPU**	**Max. Memory**
25	1.26	1.83
30	2.54	2.17
35	1.63	2.06

Table E.11: CreateCherry - TPS Rate (e2-standard-4) for Caliper

ReadCherry		
TPS	**Max. CPU**	**Max. Memory**
300	1.50	2.30
400	1.89	1.82
500	1.66	2.06

Table E.12: ReadCherry - TPS Rate (e2-standard-4) for Caliper

E.3 Machine 3 (e2-standard-8)

E.3.1 Test 1 (Workers Number)

CreateCherry		
N. of Workers	**Max. CPU**	**Max. Memory**
50	2.02	4.00
75	2.62	5.42
85	3.52	4.05

Table E.13: CreateCherry - Workers Number (e2-standard-8) for Caliper

ReadCherry		
N. of Workers	**Max. CPU**	**Max. Memory**
50	1.68	3.22
75	2.18	4.30
85	2.33	4.67

Table E.14: ReadCherry - Workers Number (e2-standard-8) for Caliper

E.3.2 Test 2 (Transactions Number)

CreateCherry		
Tx Number	**Max. CPU**	**Max. Memory**
3000	2.49	4.79
6000	2.92	5.54
10000	3.44	2.56

Table E.15: CreateCherry - Transactions Number (e2-standard-8) for Caliper

ReadCherry		
Tx Number	**Max. CPU**	**Max. Memory**
30000	2.83	5.75
60000	2.65	4.90
100000	2.33	5.88

Table E.16: ReadCherry - Transactions Number (e2-standard-8) for Caliper

E.3.3 Test 3 (TPS Rate)

CreateCherry		
TPS	**Max. CPU**	**Max. Memory**
45	2.49	4.79
50	2.92	5.54
60	3.12	5.80

Table E.17: CreateCherry - TPS Rate (e2-standard-8) for Caliper

ReadCherry		
TPS	**Max. CPU**	**Max. Memory**
500	2.83	5.75
600	2.65	4.90
700	3.02	5.92

Table E.18: ReadCherry - TPS Rate (e2-standard-8) for Caliper