João Pacheco

BSc in Computer Science

Smart Contracts using Blockchain

Dissertation submitted in partial fulfillment
of the requirements for the degree of

Master of Science in
Computer Science and Informatics Engineering

Adviser: Ricardo Nunes, Associate Manager, Novabase

Co-adviser: António Ravara, Associate Professor, DI-FCT, Universidade Nova de Lisboa

FACULDADE DE
CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

September, 2019

Smart Contracts using Blockchain

This document was created using the (pdf)LaTeX processor, based in the "novathesis" template[1], developed at the Dep. Informática of FCT-NOVA [2].
[1] https://github.com/joaomlourenco/novathesis [2] http://www.di.fct.unl.pt

Abstract

The contract is the sovereign tool employed to manage agreements between entities in today's society. It plays a crucial role in a variety of different fields, ranging from politics to finance. This fact implies the efficiency of these applications is determined in part by the efficiency of the contracts they rely on.

Despite their important role, contracts have changed relatively little in the last few centuries and remain based on an outdated technology of bureaucracy and procedures done by hand. Such systems are full of unnecessary complications, are incredibly wasteful in terms of time, money and resources, and are susceptible to human failure.

In the last few years, a type of contract represented by a computer program has appeared. This concept, known as a smart contract, is based on the emerging blockchain technology. Blockchain is a type of distributed system which assures the immutability of data via the use of mathematically secure cryptographic techniques and that, as will be discussed, is well-suited for the implementation of smart contract systems.

Transitioning contracts into the digital era would not only allow them to catch up to the technological pace of society but also would be advantageous from a safety and efficiency standpoint. This body of work will test the feasibility of using blockchain-based smart contracts to facilitate the first steps of this evolution.

This thesis assembles a proof of concept platform that supports the specification and execution of smart contracts on a blockchain network. This proof of concept will in particular target the use case of opening a bank account, aiming to create an efficient, permanent, reliable and safe process.

To achieve this, we constructed a Hyperledger Fabric network. We present herein the system developed and discuss the nuances pertaining to deploying a codebase on a blockchain, the evaluation of our system, and finally some visions for further development of this and related use cases.

Keywords: blockchain, smart contracts, Hyperledger Fabric, bank account opening, distributed systems

Resumo

O contrato é a ferramenta soberana que rege os acordos entre entidades na sociedade moderna. Desempenha um papel crucial em toda uma variedade de áreas, desde a política aos serviços financeiros. Este fato implica que a eficiência destas organizações é determinada em parte pela eficiência dos contratos de que dependem.

Apesar do seu papel fulcral, os contratos mudaram relativamente pouco nos últimos séculos e continuam assentes numa tecnologia antiquada de burocracia e procedimentos manuais. Estes sistemas contêm por vezes complicações desnecessárias, geram desperdícios de tempo, dinheiro e recursos, e estão sujeitos a falhas de origem humana.

Nos últimos anos têm sido idealizados contratos representados por programas de computador. Este conceito, conhecido como um smart contract, baseia-se numa tecnologia emergente referida como blockchain. Esta consiste num tipo de sistema distribuído que assegura a imutabilidade dos dados através de técnicas criptográficas matematicamente seguras e, como será discutido, reúne todas as condições para a execução descentralizada dos smart contracts.

A transição dos contratos para a era digital não só permitirá acompanhar o ritmo tecnológico da sociedade como trará vantagens em termos de eficiência e segurança. Neste trabalho será posta à prova a viabilidade de usar smart contracts assentes em blockchain para facilitar os primeiros passos deste evolução.

A contribuição esperada desta tese é a construção de uma prova de conceito consistindo numa plataforma que suporta a especificação e execução de smart contracts numa rede de blockchain construída usando a tecnologia . Essa prova de conceito visa particularmente aplicar-se ao caso de estudo da abertura de uma conta bancária, com o intuito de criar um processo eficiente, permanente, fiável e seguro.

Para alcançar estes objetivos, iremos entrar numa discussão em detalhe acerca da construção de uma rede Hyperledger Fabric, os pontos a ter em conta para implementar código assente numa blockchain, a avaliação do nosso sistema, e finalmente algumas visões para próximos desenvolvimentos deste caso de estudo e outros relacionados.

Palavras-chave: blockchain, smart contracts, Hyperledger Fabric, abertura de conta bancária, sistemas distribuídos

Contents

List of Figures

List of Listings

Glossary

absolute finality a certain guarantee of a transaction's immutability.

access control the set of policies that control what entities have read and/or write access to specific assets on a system.

anchor peer a special peer which, in addition to its regular functions, oversees the service discovery of other peers.

asymmetric cryptography consists in generating, for a given node, a pair of cryptographic keys: one is public and is used to interact with other nodes; the other is private and should only be known to the node.

banking core the back-end system that manages the bank's internal database and processes vital functions across all the bank's branches.

blockchain a DLT where state transitions are stored in data blocks, using cryptographic techniques to ensure the immutability of those blocks.

Byzantine fault tolerance (BFT) a system's tolerance to byzantine faults.

Byzantine fault any malicious or arbitrary network failure where a node manifests different symptoms to different communication endpoints.

Certificate Authority (CA) an administrative authority responsible for managing the set of accredited entities on a PKI distributed system by controlling and distributing the certificates that authenticate those identities.

chaincode a package of smart contracts running on a Hyperledger Fabric channel.

channel any of the media through which a customer accesses a bank's services.

clause an item in a contract that specifies some condition to be fulfilled or action to be performed.

consensus a consistent state that is identical across every correct node of a distributed data store and the process involved in achieving this state.

consensus algorithm a algorithm employed to achieve consensus.

consortium blockchain a permissioned blockchain accessed by a consortium of cooperating entities.

container an isolated virtual environment inside of which a program only has access to a narrow view of variables and resources allocated to that container and communicates exclusively with its host through a bridge.

containerisation the practice of executing a program inside a separate virtual container.

contract an enforceable agreement between two or more parties to exchange goods or services, effective immediately or during some time span.

crash fault tolerance (CFT) a system's tolerance to crash faults.

crash fault a node failure occurring when a node halts, crashes or otherwise becomes unreachable.

cryptocurrency a cryptographically-augmented, decentralized, anonymous form of virtual currency.

decentralized describes a system which lacks a central coordinating authority.

digital channel the platform containing all the services and solutions that compose online banking, most of which have been moved from the traditional branch banking experience to a mobile approach.

digital certificate a cryptographic document signed by a CA that a node uses to legitimize its ownership of a public key and, therefore, its identity.

digital signature an artifact, formed by encrypting a message using a node's private key, that authenticates that node's claim of producing the message.

distributed ledger technology (DLT) a distributed data store that maintains a state consisting of a long list of transactions.

distributed data store a database management system embedded into a distributed network which stores the data across multiple nodes.

distributed system a computing system composed of a cluster of servers that are located on different physical machines and are connected under the same network, each server performing a subset of the system's processes.

endorsement an endorser peer's declaration of having executed a given transaction.

endorser peer a peer that executes containerised chaincode and returns proposal responses containing their endorsement in the form of a digital signature.

escrow an arrangement established when two or more parties that wish to engage in a trade of money or goods bestow those assets upon a trusted third-party - the escrow agent - that distributes them accordingly.

finality the degree of immutability a ledger provides.

fork an event that spawns two separate sub-chains from a common block.

genesis block the first block of a blockchain.

hash the output a hash function produces for a given input.

hash function a function that maps a variable-size message to a fixed-size hash, deemed suitable for applications in cryptography when the relationship between input and output is chaotic.

immutable describes the past transactions of a ledger as permanent and unchangeable.

intractable describes a problem for which the computation of any solution demands too many resources to be useful in practice, even though the problem is in theory solvable via a brute-force approach.

ledger a database composed of a long well-ordered list of state changes.

miner a node that expends resources in order to generate a block's proof of work.

network gateway an interface that abstracts the communication mechanisms required for interaction with the network, provides a transparent separation of the connection scheme from the rest of the application processes.

onboarding the bank account opening process.

orderer a specialized node that does not execute chaincode but rather composes the ordering service, being in charge of packaging transactions into blocks and producing a consistent total order of those blocks.

ordering service the infrastructure that executes consensus mechanisms on a Hyperledger Fabric network.

organization each of the mutually-exclusive groups of nodes that constitute a Hyperledger Fabric network.

peer-to-peer (P2P) a distributed system where tasks are divided between functionally equal nodes - the peers - that communicate without the need for centralized coordination.

permissioned describes a closed system which can only be accessed by certified predetermined entities.

permissionless describes an open system any entity with an internet connection can maintain in the form of submitting and validating operations.

private blockchain a permissioned blockchain accessed by a single entity.

probabilistic finality a high level of merely mathematical confidence in the irreversibility of a transaction that becomes exponentially stronger the further that transaction's block is buried beneath newer blocks.

public blockchain a permissionless blockchain.

publish-subscribe a message delivery model where a cluster of brokers serves as a middle layer for the distribution of messages (classified by topic) between publishers, who produce messages, and subscribers, that consume them by subscribing to topics.

smart contract a general-purpose, though typically business-oriented, program running on a blockchain.

stake an interest in a blockchain materialized as some sort of investment.

state delta each of the atomic differences between the initial state and the state at the end of runtime, originated by the execution.

swarm a cluster of containers in a distributed system orchestrated by Docker Swarm.

third-party an independent unbiased party that mediates an activity in which it is not involved.

transaction each state changed stored on a distributed ledger.

wallet the set of digital certificates, each representing an identity or alias, of a user.

world state a materialization of the current ledger state, representing the endmost result of these changes as a collection of key-value pairs.

Chapter 1

Introduction

This chapter serves as a first contact with the information and concepts needed to understand the purpose of this thesis. The following sections will expose the reader to:

- from a business perspective, the context surrounding the contract, including its importance and the problems derived from the way it is currently handled; as well as an introduction to the smart contract, its properties and its viability
- a description of the specific business use case to be tackled and the proposed solution, from which a proof-of-concept can be derived, which can be applied to achieve the stated goals
- the underlying business setting inside Novabase and its place inside an overarching project that aims to improve the security of various use cases inside the financial services industry

1.1 Context

1.1.1 Motivation

The ongoing digital revolution has radically altered the way people interact. However, contracts - the mechanism that traditionally mediates plenty of our formal relationship - have seemingly remained unchanged ever since their inception. Due to their omnipresence and vital role in a variety of societal matters, the importance of competently transitioning contracts into the modern era is evident.

The current trajectory of the information industry is headed towards the ever-increasing integration of digital services into our day-to-day life. As we become more interconnected, questions are arising related to the trust, privacy and responsibility of the systems that govern these services. It is consequently natural for contracts to become more autonomous, safe and dependable.

This thesis explores blockchain technology, an emerging technology at the forefront of the collective endeavour that delves into these topics, and its potential to metamorphose the traditional contract into the smart contract, a digital contract regulated by a computer program running on a blockchain network. In particular, we will steer our efforts towards the use case of applying smart contracts into refining the relationship between a bank and a client established during the opening of an account.

1.1.2 Current Practices

The contract is a commonplace concept that needs no introduction; regardless, it is imperative to establish a reference definition we can work with. A contract is defined as "a binding agreement between two or more persons or parties"[1]. We will further extend this definition to encompass any enforceable agreement between two or more parties to exchange goods or services, effective immediately or during some time span.

Contracts are composed of clauses that represent conditions to be fulfilled or actions to be performed. Consenting to a contract implies an intention to satisfy all of its clauses, in addition to knowledge of the legal bond the contract enjoys i.e. unlike a promise, breach of contract gives all wronged parties the right to pursue legal action and receive reparations for incurred damages.

Currently, contracts are formed when parties reach mutual assent and physically sign the documents that comprise the contract. Some digital solutions, like filling online forms, have slowly seen adoption in recent years; nonetheless, even these typically require the involvement of trusted third-parties that assume responsibility for the process of supervising and ensuring the fulfillment of all contract clauses.

1.1.3 Problems of Conventional Contracts

1.1.3.1 Issues of Contracts in General

First of all, some clarification is required regarding the issues that this effort *does not* aim to remedy; issues that pertain to *all* contracts.

Both conventional and digital contracts suffer from the shortcomings inherent to translating real-world contexts into the technical language of legal documents. For instance, consider embedding into a contract concepts like good faith, honor or reasonability.

Additionally, like any protocol, the specification of a contract can be deformed by the individual entity that stipulates it (the same way no two programmers will implement

[1] merriam-webster.com/dictionary/contract

any non-trivial algorithm identically). Also, we can apply concerns about a contract's potential for ambiguity of meaning under certain interpretations or in certain contexts.

The current state-of-the-art of smart contracts is unable to tackle these problems because they are rooted in the very concept of the contract as we know it. In fact, smart contracts present the added challenge of translating the (already potentially flawed) legal language of contracts into a rigid programming language designed to be executed by a machine.

All of these problems are related to implementation rigor and require regulations and innovations in the study of formal verification of contract correctness, therefore falling out of the scope of this thesis. Instead, we will focus on the issues relating to efficiency and security which will be approached in the following section.

1.1.3.2 Problems of Conventional Contracts in Particular

This section musters several problems surrounding the traditional contract, which (as we will see in 1.1.4), the smart contract has the power to correct.

First, the involvement of humans in the supervision and enforcement of contracts yields inefficiency, because any changes to the contract process must be executed manually and propagate through many agents. Worse still, this dramatically exacerbates susceptibility to faults because of human error. In general, systems that are formed by few and highly automatized components perform better.

Widespread contract bureaucracy (in the sense of red tape) arises as a result of the need to coordinate complex systems comprised of many people. From the perspectives of both business and client, an incentive to streamline contract processes exists, as even marginal gains in efficiency translate to monetary profit, especially when applied at a large scale.

Often, contracts require a mediator - a third-party - that attests to the validity of the entire process. The involvement of this entity assumes an implicit relationship of trust with all parties and may require the disclosure of private information. Removing this third-party would yield better efficiency and security, in addition to abolishing the evident costs of keeping yet another component in the contract process.

For this discussion, emphasis ought to be placed on the fact that any human intervenient possesses free will. Such an agent can exhibit a malicious behaviour or act in the benefit of one or more contract parties. In fact, even a well-intentioned person can blunder due to negligence or ignorance.

1.1.4 Enter the Smart Contract

The smart contract was introduced by Nick Szabo [12] in 1996 to define a type of contract consisting of protocols that were programmed to be executed and verified by a computer in place of a human. In his definition, Szabo identifies some properties that set it apart from the simpler digital contract:

1. the contract is governed by a machine instead of a person
2. all changes to the state of the contract are credible without a third-party
3. all transactions are saved in a manner that is irreversible (or that at the very least is mathematically infeasible way to revert in practice)
4. performance (or violation) of the contract is observable transparently by all parties
5. the contract is enforced deterministically without interference

Whilst smart contracts are not exclusively dependent on blockchain, the approach of placing the computation of digital contracts on a blockchain network is currently the best candidate to satisfy the above requirements. In fact, the smart contract soon became linked to blockchain technology and eventually morphed into the idea of any (though typically business-related) general-purpose program that runs on a blockchain. For this thesis, the term will henceforth be used to refer to this more modern definition that implicitly assumes the involvement of blockchain technology.

Regarding the properties enumerated above, the smart contract stands out as clearly superior to the conventional contract. Being a deterministic automaton devoid of free will, it secures reliability because operations are categorically performed exactly as stipulated, in the expected order and without interference from a third-party. Running on a computer means that execution and event propagation are as efficient as the underlying technology allows it. Finally, the smart contract enjoys the security advantages of blockchain, namely the immutability of transactions.

1.2 Problem & Solution

1.2.1 Problem Description

The problems of conventional contracts affect all industries where most procedures are to some extent shaped by contracts. In a financial services context, plenty such use cases are available for this thesis to tackle. The one we will focus on is the opening of a bank account, which entails the formalization of a relationship between the bank and one or more clients[2]. Reliance on conventional contracts results in a mediocre account opening process characterised by the three key problems discussed previously: inefficiency, red tape and the necessary presence of third-parties.

Digital contracts, used in online banking, are a step in the right direction to combat these pitfalls. However, unlike their physical siblings, digital contracts can be tampered or become corrupt. Their mutability leaves an unwanted vacuum - in the eyes of the bank - in the relationship with the client during the interregnum between the moment an account creation is requested and the moment it is officially created, especially in the

[2]This document assumes one client (*singular*) for simplicity, unless noted.

case of requests by geographically-distant customers. Because of this, banks are hesitant to trust digital contract solutions, which constitutes a major hurdle to the widespread adoption of online banking.

In short, we are attempting to solve the following predicament:

A period of uncertainty in the relationship between a bank and a client exists between the account request and the activation of the contract.

1.2.2 Proposed Solution

The process of opening a bank account begins with an initial contact where a client communicates their intention to a bank. Assuming that all prerequisites (for instance agreement with terms of use) are met, the account is registered. At this point, although the account is in place, it exists in a dormant state until certain contract clauses are fulfilled. In our particular use case, the contract must be ratified in the form of a signature by the following three individuals: the client, the account manager and a bank representative.

The goal of this thesis is to create a system of smart contracts embedded in a blockchain which a bank can use as a tool to better manage its relationship with a client in the intermediate time between forming and putting into effect a contract. By storing all interactions and changes to the process on a blockchain, the smart contract acts as an immutable bond established between the two entities, both as an intermediate connection between the parties and as a perpetual guarantee of the authenticity of the process.

The following sequence of diagrams illustrates, from a macroscopic and business-centric perspective, the flow of the use case when the proposed solution is integrated:

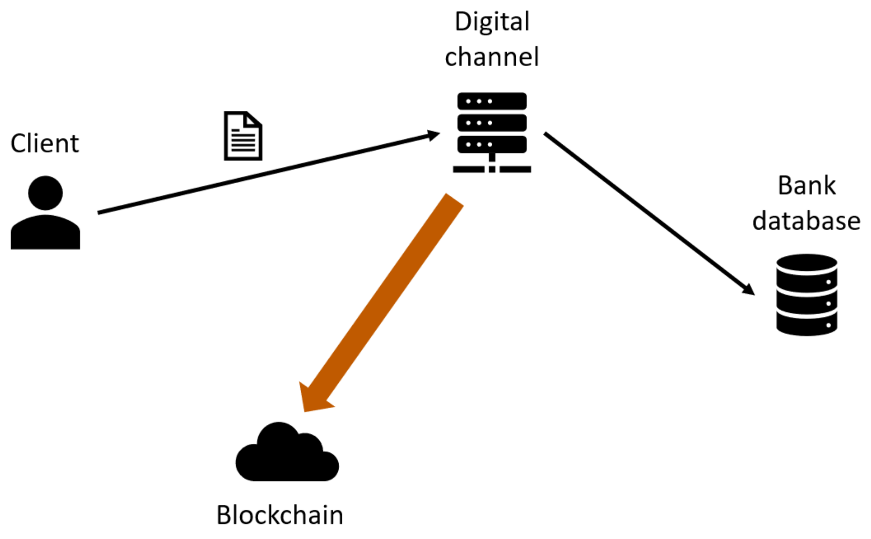


Figure 1.1: The client expresses an intention to open a bank account by submitting a request to the digital channel, which creates it in the internal bank database. Simultaneously, the digital channel registers the account as a smart contract on the blockchain.

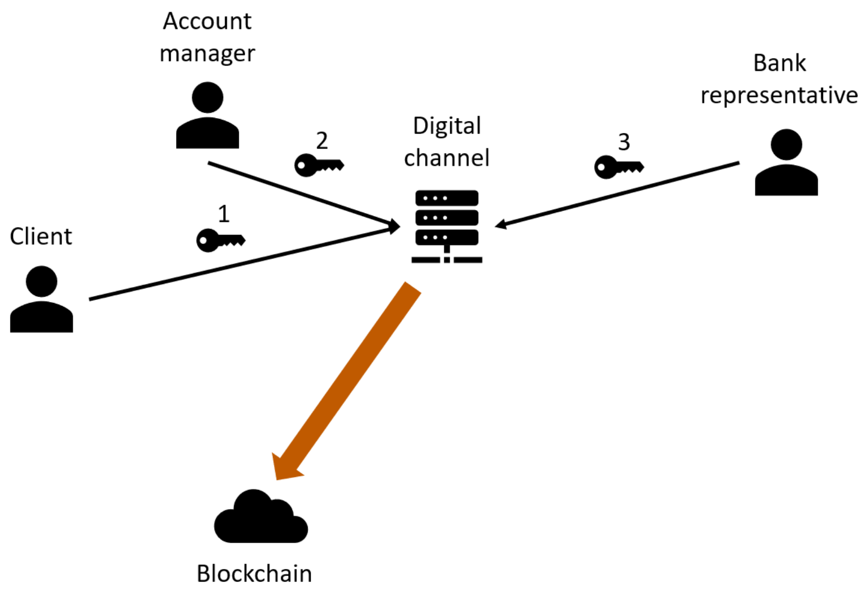


Figure 1.2: The smart contract rests in a period of signature gathering, where it can either be activated or annulled. It awaits incoming signature submissions from the digital channel until all contract clauses are met.

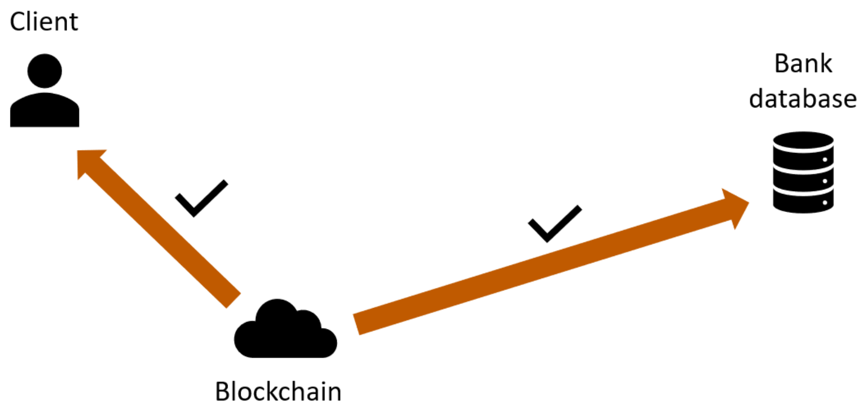


Figure 1.3: Once all signatures are present, an event is fired that propagates a notification about the activation of the account to both the client and the database.

1.3 Project Setting

The proof-of-concept described in this document was developed in a business environment inside Novabase. In the near 30 years since its establishment, Novabase has become the leading Portuguese IT company and is listed in the Euronext Lisbon stock exchange since 2000. It has installations in Spain, Belgium, the United Arab Emirates, Mozambique, Angola, the United Kingdom, Turkey and Portugal, totalling more than 2000 collaborators. It currently provides IT solutions for the financial services, government, healthcare, energy, utilities and aerospace sectors.

This proof-of-concept, among other endeavours in emerging technologies like blockchain, is part of an internal project called Safehub. The project aims to create a platform that facilitates the digital authentication of contract documents. Being co-financed by the European Union, it also requires compliance with the laws of the EU sphere.

All entities that offer financial services require the means to authenticate a client as well as provide safe methods of forming contracts, both in a simple and friendly way yet under the assurance of high security. Novabase's mission for this thesis is the study of the applicability of blockchain technology to business use cases in the financial services industry to achieve one of the objectives of the EU single market.

Chapter 2

Technical Background

The purpose of this chapter is to acquaint the reader with several concepts that make up the technical background of this thesis, which belong to the fields of distributed systems, cryptography and blockchain.

Afterward, we will present the research that was done into consensus algorithms and the rationale (based on this research) behind the choice of Hyperledger Fabric as the framework on which to build the proof-of-concept discussed in the last chapter.

2.1 Distributed Systems

A distributed system[5] is a computing system composed of a cluster of servers that are located on different physical machines and are connected under the same network. Each server performs a subset of the system's processes and must communicate and synchronize with other servers.

Such a system can be deployed to carry out particularly complex and demanding tasks that cannot be performed in viable time using a single-computer setup. Even for solving comparatively effortless problems, a distributed system leverages load balancing to attain dramatically improved performance. Finally, a distributed architecture is also designed to eliminate single points of failure and bottlenecks, problems that can bring an application to a halt.

The study of distributed systems is a vast field of computer science, encompassing a handful of concepts of which comprehension is vital for understanding the rest of this thesis.

2.1.1 Distributed Data Store

A distributed data store[10] is, put simply, a database management system embedded into a distributed network that stores the data across multiple nodes. Database replication allows for the advantages of distributed systems to be applied to conventional databases. On the one hand, it allows the distribution of system loads by spreading operations across an array of servers that share the same data. On the other hand, it aims to achieve some level of fault tolerance by allowing a system to recover from a serious failure via one or more up-to-date copies of the database state prior to the fault.

A distributed database across a cluster of servers can be stored in one of two ways:

Replication
Redundantly maintaining identical copies of the data across a cluster of different servers, allowing for operations to be dispersed across equally capable servers. Replication also offers ideal safety, because a data back-up can be recovered from any replica. This comes at the expense of increased data storage capacity demands. Moreover, a replicated system requires more complicated logic to deal with the higher potential for data conflicts between replicas.

Partitioning
Dividing the database into distinct parcels, each of which is allotted to a different node, saving total disk storage needs. Data partitions can be completely disjoint, at the risk of permanent loss of data should one node fail. A safer approach is introducing some redundancy or overlap between partitions so that data can be restored from one or more other nodes in the event of a crash.

Even though data is mirrored or split across many access points, proper distributed data stores are transparently abstracted by the overlying distributed system such that they are viewed as a unitary database from an external observer. In other words, a client should interact with a replicated database as though it were a regular monolithic database when, in reality, it is a shared state.

2.1.2 Peer-to-Peer

In a peer-to-peer (P2P)[11] system, tasks are divided between functionally equal nodes - the peers - that communicate without the need for centralized coordination. Peers have equivalent responsibilities and perform both read and write operations using their own data storage and processing resources. This contrasts with the traditional client-server model, where client nodes exclusively request data from server nodes that execute back end operations.

Peer-to-peer systems are more scalable and robust than client-server networks, since every node in a peer-to-peer network is a client capable of carrying out identical operations, thus sharing its resources but being somewhat replaceable. Peer-to-peer technology is most commonly employed to build decentralized file-sharing systems.

2.1.3 Fault Tolerance

Distributed data store processes can be disrupted by two types of node failures. On the one hand, a crash fault occurs when a node halts, crashes or otherwise becomes unreachable. It should be noted that all messages to and from an affected node are dropped, regardless of the observer. Byzantine faults extend the concept of crash faults and encompass any malicious or arbitrary network failure where a node manifests different symptoms to different communication endpoints, resulting in a network environment where nodes cannot fully trust each other. Such faults are illustrated by the Byzantine Generals Problem[8], which describes the analogy of a group of generals that must decide on a common plan bearing in mind the fact that several of them might be unreliable.

A system that can continue to correctly function despite the occurrence of faults is said to, depending on the type of fault, exhibit either crash fault tolerance (CFT) or Byzantine fault tolerance (BFT).

2.1.4 Consensus

As we have surmised, a distributed data store maintains a single state across multiple geographically separated nodes. This premise introduces the challenge, foreign to single-node systems, of ensuring that every node agrees on the state that is managed. A data system is said to achieve consensus if (within some margin of latency) it guarantees a uniform state that is consistent with the operations directed at any correct node[5].

The nuances of consensus and the related protocols in the context of blockchain will be explored thoroughly in section 2.4.

2.2 Cryptography

2.2.1 Digital Signatures

Asymmetric cryptography[6] consists of generating, for a given node, a pair of cryptographic keys: one is public and is used to interact with other nodes; the other is private and should only be known to the node. This form of cryptography is typically used to transmit an encrypted message between two nodes: some node A can encrypt a message's contents using some node B's public key, who can then decrypt it using its private key and obtain the message in plain text.

A digital signature is formed when a node A encrypts a message with its private key. If A wants to authenticate itself as the sender of the message to some node B, it can provide the digital signature along with the original message, at which point B can decrypt using A's public key and (hopefully) obtain a result identical to the message plain text.

Each node on a distributed system is assigned a unique asymmetric cryptography key pair, the private key of which must be kept secure. To perform operations, a node needs to authenticate itself in the form of a digital signature. Unless compromised, the confidentiality of a node's private key ensures that the network is protected against masqueraders attempting to gain access under the guise of that node. On the flip side, digital signatures also allow the receiver of a message to refute attempts by the sender to repudiate ever sending the message.

While digital signatures are the engine of entity authentication (and the subsequent possibilities for access control) on a network, blockchain's pivotal property is the immutability of data, which is achieved with the use of cryptographic hash functions.

2.2.2 Hash Functions

A hash function[6] maps a variable-size input to a fixed-size output (the hash, also called "message digest"). Some hash functions are deemed suitable for applications in cryptography, notably when the relationship between input and output is chaotic. This means that a proper cryptographic hash function produces completely and unpredictably different hashes for messages with even minute changes between them. Also, an apt cryptographic hash function must compute in significantly faster times than a symmetric-key encryption algorithm (for equal-size inputs), otherwise it would offer no benefit over the latter. For the purposes of cryptography, this discussion only considers deterministic functions i.e. functions where if we use the same message as input we will always obtain the same output.

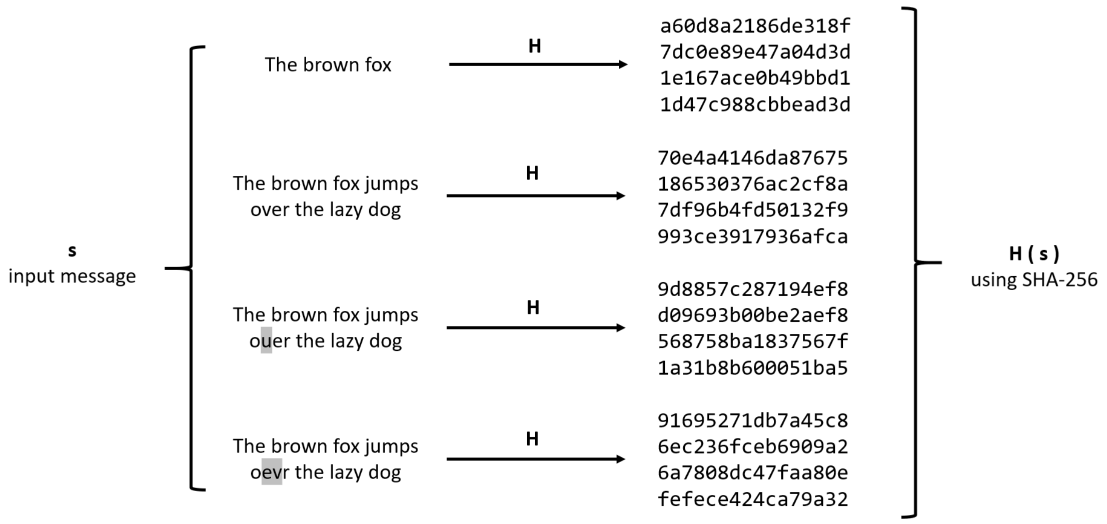


Figure 2.1: Slight variations on the same input result in radically different hash values.

Because a hash function translates an arbitrarily big space of messages into a limited set of possibilities, the possibility for a hash collision to occur between any two inputs exists. To mitigate this, a well-designed hash function produces output hashes with a sufficient number of bits such that its codomain is incredibly big. For instance, a 256-bit output space contains $2^{256} \approx 10^{77}$ possible hash values (for comparison, estimates place the amount of matter in the observable universe at approximately 10^{80} atoms). At this scale, the probability of a collision is *extremely* low.

Calculating the hash for some arbitrary input value is easy; however, calculating what input was fed into a hash function to produce a certain output hash is described as an intractable problem. This entails the computation of any solution demands too many resources (is infeasible) to be useful in practice, even though the problem is, in theory, solvable via a brute-force approach. Likewise, it is intractable to procure two messages with the same hash. Finally, for a proper cryptographic hash function, it is impossible to learn anything about some input by looking at its corresponding hash (and vice-versa).

2.3 Blockchain

In popular culture, blockchain is typically equated with cryptocurrency, a form of cryptographically augmented, decentralized, anonymous virtual currency. Most notably, it is used interchangeably with the cryptocurrency Bitcoin, which was proposed and then implemented by Satoshi Nakamoto [9] in 2009, and over time became the flagship for both cryptocurrencies and blockchain in general.

Albeit an interesting and useful use case, this document employs the term to refer to what also commonly, but perhaps more accurately, is described as a distributed ledger technology (DLT): a type of distributed system with some special characteristics that will be thoroughly explored in this chapter. Nevertheless, Bitcoin implementation details are sparsely mentioned to help illustrate some points.

Succinctly, blockchain[3] is a network, specifically a distributed system composed of geographically dispersed participants. Nodes are arranged in a peer-to-peer architecture and are tasked with replicating a shared state - the ledger - through consensus mechanisms. The ledger is continually built upon by the nodes using cryptographic protocols that ensure the immutability of all information that is added to it.

2.3.1 Ledger

The ledger[3] is a database consisting of a long history of state changes. These changes are called transactions and represent operations acted upon objects on the blockchain. For example, on the Bitcoin system, the ledger records money transfers between participants. Incoming transactions are grouped into batches that are added to the ledger in chronological sequence, forming a long chain of data blocks (hence the name blockchain).

We will represent the first block on a blockchain - called the genesis block - as b_0. When the genesis block is stored on the ledger, a hash of its contents $H(b_0)$ is stored alongside it. The next block b_1 points to this hash by including it inside its contents before it itself is hashed. Then, the newly generated $H\Big(\ b_1 \parallel H(b_0)\ \Big)$ is stored alongside b_1 on the ledger. Every valid block b_i that is subsequently added to the ledger must continue this pattern of concatenating the hash of the previous block to its own contents and saving the result of the operation $H\Big(\ b_i \parallel H(b_{i-1})\ \Big)$ to the ledger.

Because of the particular properties of cryptographic hashes, a unidirectional dependence $H(b_i) \leftarrow H(b_{i+1})$ exists, forming a cryptographic link between every block and its successor. Because every block is linked to the one immediately before and referenced by the one immediately after, a chain of connected blocks emerges that stretches all the way to the genesis block. Each link $(b_i,\ b_{i+1})$ in the chain corroborates the integrity of b_i, which itself confirms b_{i-1}, and so forth.

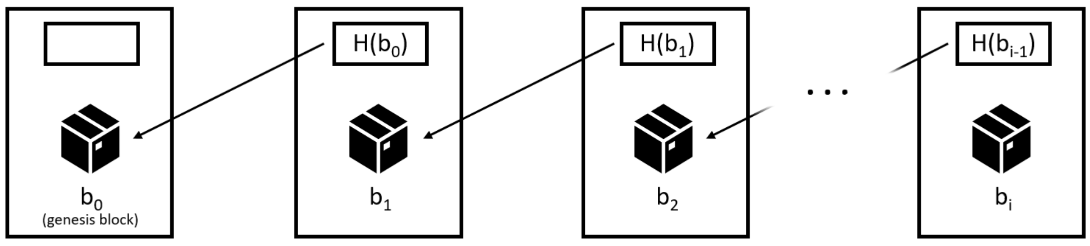


Figure 2.2: Every block influences the hash value of the next, creating a chain of blocks.

2.3.2 Immutability

The ledger is immutable because, even though technically possible, blockchain demonstrates a remarkably strong resilience to database tampering attacks[3]. In practice, altering a local ledger unnoticed also requires faking a considerable portion of the chain links, while perverting the global ledger is impossible without first obtaining supremacy over the network.

A valid block's hash must be congruent with its contents. Any corruption of the block's data or hash results in a discrepancy between the block's declared and expected hashes. A correct node will compute the hash of every new block it receives and reject it if such a discrepancy is detected.

Likewise, it is equally trivial to discern a posteriori modifications to any block already on the ledger, because of the relationship of its hash with the next block's. Suppose an attacker manipulates some block b_i such that it also states a new hash $H'(b_i)$ that corresponds with the falsified data. An examination of the next block will reveal that its hash $H(b_{i+1})$ does not match the result of the operation $H\Big(\ b_{i+1} \parallel H'(b_i)\Big)$ using the adulterated hash.

Because of the aforementioned $H(b_i) \leftarrow H(b_{i+1})$ relationship for any pair of cryptographically linked blocks, even if the attacker also forged $H'(b_{i+1})$, an inspection of b_{i+2} would result in the same finding. In fact, to inconspicuously modify a single block b_i an attacker must forge the hash of every block b_{i+1} through b_I accordingly, where I is the total number of blocks on the chain.

Any node can, even if at the risk of an audit, produce such major changes to its own copy of the ledger. However, to alter the global ledger, an offender would have to propagate the fake blocks to the rest of the network. In a blockchain where correct nodes retain control over the consensus layer, the blocks would then be swiftly discarded by every benign node. Therefore, to force even the slightest change to an existing block, an attacker needs to control enough nodes to overpower the consensus layer (which depending on the consensus algorithm might imply a 51% quorum).

2.3.3 Access control

Blockchain networks can be classified according to their type of access control i.e. the *base* level of permissions that is required to interact with the ledger[3]. Although this descriptor merely states whether operations are readily accessible to the open public, access control type has a deep influence on the consensus mechanisms of a blockchain.

Public blockchains are open repositories where any entity with an Internet connection can become a node capable of reading the ledger, as well as submitting and validating transactions. Because they lack any kind of access control, public blockchains are also designated permissionless. Public blockchains are considered fully decentralized because they lack any form of central authority. Instead, governance over the ledger is distributed across the peers[1]. Decentralization allows for a trustless paradigm in which the necessity of network members to trust each other is kept at a minimum.

Conversely, private blockchains are described as permissioned because access is only permitted to participants who have been vetted by an administrator and granted membership. Some incarnations of this model may, however, allow public access to read operations and other relatively low-stakes functions. We can further distinguish purely private blockchains from consortium blockchains, in which the network is governed by a partnership of multiple independent but cooperating entities, as opposed to a single individual or institution. In either case, because any form of administration evidently constitutes a central authority, permissioned blockchains are considered centralized.

2.4 Consensus Algorithms

In the context of blockchain, consensus can be defined as the state reached when all correct processes act upon an identical ledger, even in the case of anomalies that affect

[1]Before the extension of the term to cover what we refer to as DLT, blockchain was limited to its public side. In fact, decentralization was originally, along with immutability of data, one of the key aspects of blockchain technology.

some subset of faulty - potentially malicious - nodes. As a type of replicated distributed data store, blockchain depends on the agreement of all correct nodes about every block that is added to the chain and in what order.

The protocols that achieve this agreement are called consensus algorithms and are an important piece of blockchain technology[3]. A plethora of blockchain-applicable consensus algorithms exists, some of which vary radically depending on the implementation. This section compares in broad strokes the consensus algorithms that are used in permissionless versus permissioned blockchains, and their implications.

Most importantly, it is permeated by consideration about the trade-off between scalability and finality i.e. how confidently we can assert that any submitted transaction is permanently stored on the ledger.

2.4.1 Permissionless Blockchain

Because public blockchains are decentralized, they must be maintained by independent users that perform peer operations. Additionally, as the network scales, even more peers are required to support the added workload. Because these users expend their own resources, some sort of subsidy is expected as an incentive for their labour. Otherwise, a permissionless blockchain cannot be expected to be sustainable (let alone profitable) because it will be unable to keep existing users and lure new ones. This incentive takes the shape of a token, the aforementioned cryptocurrency, that represents an investment on the network and can be exchanged for goods in markets that accept it. For the above reasons, from here on, this document draws an equivalence between *viable* public blockchains and cryptocurrencies.

Public blockchains employ lottery-based consensus algorithms that pick an individual peer to add each block (herein referred to simply as "minting" a block) and receive a reward in the form of cryptocurrency tokens. We will introduce two public blockchain consensus algorithms as examples:

Proof of Work

Used in Bitcoin, Proof of Work (PoW)[9] requires peers, which are called the miners, to expend a non-trivial amount of processing resources solving a puzzle to be eligible to mint a block. Miners must find some value n such that $H\Big(\ n \parallel H(b)\ \Big) < k$, for some predetermined k. In other words, they must iteratively test integers until a number is found that, when concatenated with the block b's hash and the subsequent result is fed into a hash function as input, produces a hash value that is smaller than some parameter k. The first peer to submit the block along with a suitable value n is selected as that block's minter.

Proof of Stake

Pioneered by the cryptocurrency Peercoin[7], Proof of Stake (PoS) is a more recent algorithm that selects block minters through a process that favours peers that have evidenced their interest in the network through some sort of investment (its stake). The simplest PoS model quantifies each node's stake as the amount of tokens that node owns and accordingly distributes minting chance in proportion to wealth[2]. PoS systems avoid the excessive resource consumption demanded by PoW but might offer weaker finality guarantees.

Lottery protocols select winners according to a distribution of power that involves random chance. In the case of PoW, peers with more computing capacity have a better chance of becoming minters; for PoS, wealthier peers have the edge. If we assume the network is composed of a majority of honest nodes, lottery-based strategies statistically benefit this majority and, in turn, the whole network.

Permissionless blockchain consensus algorithms must also ensure the validity of transactions in a trustless environment where membership is not restricted. To achieve this, they are designed around assumptions about the behaviour of the most trustworthy participating nodes, such that power is steered towards those nodes. PoW is built around the hypothesis that the agents that invest most resources into mining blocks are the most committed to the integrity of the network; PoS reasons that the wealthiest nodes are the most faithful because they have the most to lose from a disturbance to the network.

2.4.1.1 Probabilistic Finality

Due to the public and decentralized context, consensus algorithms in permissionless blockchains must be viable on a potentially global system. They handle scalability concerns at the expense of ensuring finality.

On public blockchains, often two or more nodes concurrently mint blocks that occupy the same position on the chain, spawning a fork. After this event, a dispute over the last index ensues where chunks of the network follow separate sub-chains, temporarily breaking consensus. The fork is only settled once the whole network designates one sub-chain as canonical and resumes adding blocks. The mechanics of fork resolution mean however that a block can be added to the blockchain, only for it (and its transactions) to be displaced by a block from a different sub-chain.

Permissionless blockchain merely provides probabilistic finality i.e. a high level of mathematical confidence in the irreversibility of a transaction that becomes exponentially stronger the further that transaction's block is buried beneath newer blocks. For example, a Bitcoin block is considered canonical once it is confirmed by six following blocks.

2.4.2 Permissioned Blockchain

Unlike public blockchains, permissioned blockchains are not built around the goal of global decentralized infrastructure. Rather, the regulated nature of their membership

[2]A cryptocurrency under this model will automatically converge towards an unbalanced environment where rich nodes amass ever-increasing wealth and control over the network. Consequently, typical PoS implementations use more sophisticated definitions of stake.

and scale reflects a shift in focus towards the quality of the services they provide. Another important aspect that sets them apart from permissionless blockchains is the lack of an incentive or reward system, as they are centralized systems hosted by private entities.

In fact, besides having a ledger, permissioned blockchains are akin to typical distributed data stores in terms of internal operation. As a consequence, they employ similar voting-based consensus algorithms that require the stable consensus of a quorum of nodes.

Practical Byzantine Fault Tolerance (pBFT)

pBFT[4] is a message-heavy protocol that simultaneously accomplishes asynchronous consensus and constitutes one possible implementation of Byzantine fault tolerant consensus. It separates the consensus process into rounds (also called views), each of which is managed by a consensus leader and, if successful, concludes with the insertion of a block into the chain. Each round, one node is designated as the leader (or primary node), while every other is a secondary node. Each round is divided into the following 4 phases:

1. transactions are transmitted to the leader
2. the leader announces the next ledger block b
3. each node validates b and broadcasts a confirmation (or rejection) response
4. if a node receives $f + 1$ confirmations, it adds b to its ledger

The protocol guarantees that, in all systems constituted of $3f + 1$ nodes, consensus can be reached in a Byzantine fault tolerant fashion if at most f nodes are faulty in any given consensus round.

Apache Kafka

Apache Kafka is a popular message brokering engine that implements consensus in a distributed data store using a publish-subscribe model. Publishers push transactions to one or more topics (typically a single topic that represents the ledger), while subscribers arrange to automatically receive transactions from the topics they subscribe to. Peers assume both the roles of publisher and subscriber, and package the transactions they receive into blocks.

Acting as mediators, brokers manage the topics, holding incoming transactions until they can be distributed to all the pertinent subscribers. Brokers additionally maintain a state of the network topology which is updated every time a node joins or leaves. This allows a loose-coupling environment in which both producers and subscribers must only establish a connection to the brokers in order to propagate and consume transactions, simplifying the interactions between them.

Kafka is, as of writing, the predominant consensus algorithm used in Hyperledger Fabric. Unfortunately, Kafka is not compliant with BFT, merely providing resilience against crash faults. Although the implementation of a BFT consensus algorithm is planned, such a module has not been released yet. We will examine this technology much more closely in the following chapter.

2.4.2.1 Absolute Finality

Voting-based consensus circumvents the processes that lead to the occurrence of forks in lottery algorithms. Namely, every peer independently manages its own ledger and engages in a strictly asynchronous consensus. As we discussed in 2.4.1, such constraints are necessary to meet the scalability needs of public blockchains, given their global scope. Instead, voting algorithms require an explicit unison before every block is added and rely on system-wide communication. As a result, a block (and its transactions) can never be overridden after being included in the ledger, ensuring the property of absolute finality.

In order to guarantee finality, these algorithms sacrifice scalability, because the addition of each block requires the transmission of a volume of messages that grows exponentially with the size of the network. However, transaction finality is non-negotiable in the business use cases of permissioned blockchains, whilst scalability is not as critical in a system with restricted membership.

2.5 Hyperledger Fabric

Hyperledger is a conglomerate of various open-source frameworks and tools related to blockchain technology applied to business use cases. One of these is Hyperledger Fabric[1, 2], a project that aims to provide the infrastructure for the execution of smart contracts - which in the Fabric vocabulary is called chaincode - on a permissioned blockchain. A Fabric network consists of peers grouped into organizations ("orgs" for short), allowing the creation of both private and consortium blockchains.

Its primary selling point is a modular and highly customizable architecture capable of adapting to the flexibility, robustness and scalability needs of any particular system. Fabric networks are configurable through the use of pluggable modules provided by the project which can also be implemented by the user. For instance, as of writing, support is available for writing chaincode in Go, JavaScript and Java.

The chaincode execution layer runs smart contracts inside virtual containers using Docker. A container is an isolated virtual environment inside of which a program only has access to a narrow view of variables and resources allocated to that container and communicates exclusively with its host through a bridge. This practice, called containerisation, is a security measure that ensures any potentially harmful effects of a contained program - in this case, a smart contract - are safely averted and cannot leak onto the parent system.

Invocations to chaincode functions are intercepted by a middle layer (called the contract interpreter) that assesses the validity of the request before executing the chaincode inside a Docker container. If the request is valid and the program runs correctly, the interpreter outputs a signed ratification (an endorsement) of the request and the set of state deltas, i.e. the difference between the initial state and the state at the end of runtime, originated by the execution. The output package is then submitted as a transaction proposal to the consensus mechanisms which involve a final validation of its signatures and state deltas.

Consensus is a complex process implemented by an infrastructure of specialized nodes called the ordering service. These nodes, called the orderers, are in charge of packaging transactions into blocks and producing a consistent total order of blocks. The consensus implementation itself is pluggable, meaning the ordering service can perform several consensus algorithms. Currently, Hyperledger Fabric provides the choice between Kafka, which uses Apache Kafka, and Raft. Raft is a recent addition to Fabric's module catalogue that unfortunately was not available as of the beginning of this thesis; therefore Raft will not be further discussed beyond this brief mention.

Every peer holds, in addition to the ledger proper, a world state, which consists of a materialization of the current data store state in a non-relational database. While the ledger forms a long string of the aforementioned state deltas, the world state represents the endmost result of these changes as a collection of key-value pairs. The world state allows chaincode to efficiently consult the state of an object without runtime traversal of the entire blockchain data structure. Hyperledger Fabric currently allows two world state database modules: LevelDB, a simpler option available natively, and CouchDB, a more sophisticated implementation optimized for complex JSON records.

Fabric's access control model incorporates a simplified Public Key Infrastructure (PKI) architecture. To participate in a network, a node requires an authenticating digital certificate. This consists of a cryptographic document signed by an administrative trusted party, called the Certificate Authority (CA), that a node uses to legitimize its ownership of a public key and, therefore, its identity. The CA is responsible for creating, distributing, storing and revoking every certificate on the network. The CA also manages the set of accredited entities on the blockchain by controlling and distributing access permission policies.

Hyperledger Fabric allows the creation and configuration of independent channels inside a common blockchain. Channels act like private ledgers that are accessible only to specific organizations and have their own individual set of deployed chaincodes. Furthermore, different channels can have entirely different configurations from one another. A Fabric blockchain can contain multiple coexisting channels and an organization can have access to many of those channels.

2.6 Choice of Technology

From our review of the nuances of permissionless and permissioned blockchain, the use case in question can be surmised to require a private solution. The business setting places the finality of transactions as a top priority that the current state-of-the-art of public blockchain consensus algorithms does not ensure. Data privacy concerns automatically exclude permissionless blockchain architecture.

Among permissioned blockchain technologies, Hyperledger Fabric stands out as the open-source framework of choice. Since the project started in 2016, its codebase and surrounding community have matured enough such that it has served as the basis for numerous proofs-of-concept in a wide variety of industries. Finally, its modular approach to blockchain allows a flexible development process.

CHAPTER 3

Blockchain using Hyperledger Fabric

This chapter details the subtleties of constructing a Hyperledger Fabric blockchain network, a chaincode codebase designed to run on said network and a client application to interact with the chaincode, complete with samples of the source code that was developed for every step.

3.1 Network Model

As part of the Safehub project, the blockchain that was developed encompasses the objectives of two proofs-of-concept in a single platform, as a means of streamlining configuration tasks and promoting the exchange of ideas. Specifically, an experiment into Know Your Customer (KYC) processes was integrated into the same network. To differentiate between the two, this proof-of-concept is referred throughout the repository (notably in directory names) as some variation on "Smart Contracts" while the other is aptly named "KYC".

Leveraging the multi-organization blockchain and channel creation capabilities of Hyperledger Fabric, our approach was to design a consortium blockchain comprising three organizations that engage in both use cases using designated channels. The three orgs were baptised with the names "B1", "B2" and "B3", representing banks that coexist on a common blockchain.

Transactions are logically separated according to the channel, and respective use case, they target. On the one hand, KYC is a use case with a global scope, involving the cooperation between multiple organizations. Thus, a single instance of KYC is deployed on a global channel, "channel-all", that is shared between orgs. On the other hand, transactions related to the digital contract use case must be performed confidentially on private sub-ledgers. To this end, Smart Contracts is deployed on three channels, "channel-b1", "channel-b2" and "channel-b3"; each of which is accessible only to its respective org.

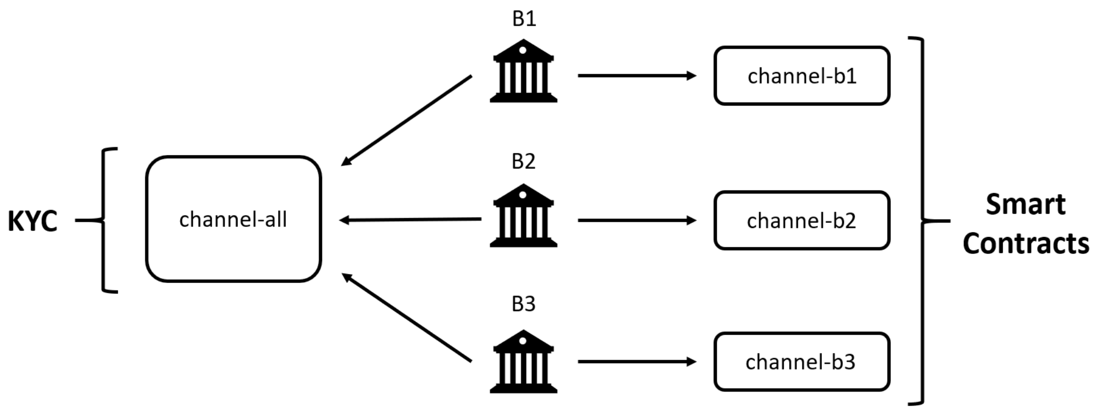


Figure 3.1: The network comprises three organizations that interact with the global channel. Each org additionally has its own separate channel.

Two peers, "peer0" and "peer1", each representing a branch, constitute each of the three banks. At least one of them is assigned the role of anchor peer, a special peer which, in addition to its regular functions, oversees the service discovery of other peers. Anchor peers learn about every peer in the network and, when contacted by other peers, circulate their knowledge. This allows the efficient propagation of updates to the view of the network membership without the need for flooding. In our system, the peer0 of each org was chosen as its anchor peer.

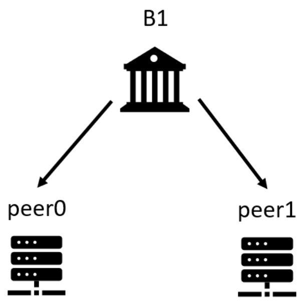


Figure 3.2: Each organization has two peers and is responsible for an orderer.

The ordering service comprises three orderers and can be technically considered organization-agnostic since orderers do not carry out transactions. As we discussed earlier in 2.5, the consensus model uses a crash fault-tolerant system consisting of Kafka servers.

3.2 Environment

A work environment was set up for this project on a virtual machine running Ubuntu 18.04 LTS. We chose the most recent (as of April, 2019) official release of Hyperledger

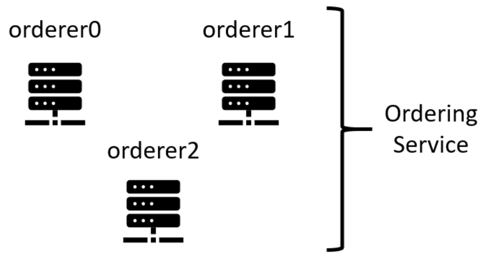


Figure 3.3: The ordering service is composed of three orderers that perform consensus mechanisms.

Fabric, 1.4.1. The following dependencies of Hyperledger Fabric were also installed:

Docker, Docker Compose
: Hyperledger Fabric peers and smart contracts are run as Docker containers hosted by the operating system. Although not strictly necessary, Docker Compose is a very helpful tool that allows the developer to define a network as a set of extensible services in one or more files, and then deploy the entire network in a single command.

Go
: Fabric is written in the Go programming language and thus requires its installation to run.

Node.js, npm
: To write chaincode in JavaScript using Fabric's SDK for Node.js, this must be installed, along with the package manager npm.

3.3 Network Construction

The first step to build a local network is to configure two YAML files which are read by binaries provided by Hyperledger Fabric. The specification inside these files is used to create a set of objects which are loaded by the network upon bootstrap.

3.3.1 Certificate Generation

As we know from 2.5, CA servers manage the digital certificates that identify all the various entities on a running Fabric blockchain. However, in order to build such a network, an initial set of these certificates must be present at bootstrap time.

Hyperledger Fabric provides an executable tool called "cryptogen" which generates the needed digital certificates and other cryptographic material. This binary reads a YAML file ("crypto-config.yaml") which details the peer and orderer orgs that initially compose the network. It then constructs a vague topology of the network and generates cryptographic material for each participating entity.

```yaml
PeerOrgs:
  - Name: B1
    Domain: b1.poc.com
    EnableNodeOUs: true
    Template:
      Count: 2
    Users:
      Count: 2
```

Listing 1: Detailing a peer org's topology

Inside crypto-config.yaml, the `PeerOrgs` array specifies the peer organizations. Special attention should be directed at the fields `Template.Count` and `Users.Count`, which respectively specify how many peers and how many user accounts (alongside an implicit admin account) the org has. The example 1 specifies the peer org B1 with namespace "b1.poc.com", two peers and two users.

```yaml
OrdererOrgs:
  - Name: Orderers
    Domain: poc.com
    Template:
      Count: 3
```

Listing 2: Detailing an orderer org's topology

On the other hand, orderer orgs don't have users and their configuration (as seen in 2) is shorter. Here, `Template.Count` specifies the number of orderers instead of peers. Since we consider the orderers as organization-agnostic, we established the sole orderer org's namespace as simply "poc.com".

The executable is run using the following command (3):

```
cryptogen generate --config=./crypto-config.yaml
```

Listing 3: Generating cryptographic material

3.3.2 Channel Artifact Generation

The second file that must be configured is called "configtx.yaml" and defines configuration profiles the user can select to create a number of artifacts. Specifically, these are:

- "genesis.block" - the genesis block used to initialize the ordering service and, therefore, the blockchain as a whole

- "$CHANNEL/generate.tx" - the generation transaction of each channel
- "$CHANNEL/anchor_peers_$ORG.tx" - for each channel, the transaction that specifies each organization's anchor peers

The file is logically separated into multiple blocks that are referenced by the profiles with YAML syntax.

```
Organizations:
    - &B1
        Name: B1
        ID: b1-msp
        MSPDir: crypto-config/peerOrganizations/b1.poc.com/msp
        AnchorPeers:
            - Host: peer0.b1.poc.com
              Port: 7051
        Policies:
            Readers:
                Type: Signature
                Rule: "OR('b1-msp.admin', 'b1-msp.peer', 'b1-msp.client')"
            Writers:
                Type: Signature
                Rule: "OR('b1-msp.admin', 'b1-msp.client')"
            Admins:
                Type: Signature
                Rule: "OR('b1-msp.admin')"
```

Listing 4: Configuring an organization's bootstrap information

Every organization is configured as its own anchor inside the `Organizations` configuration block we see in 4. By convention, the set of certificates that represent the identity of the various entities of an org are abstracted as a Membership Service Provider (MSP). Here, `MSPDir` is the location of these certificates. Also relevant are `AnchorPeers`, an array which defines all the org's anchor peers, and `Policies`, the organization's configuration management policies.

```
Orderer: &OrdererDefaults
    OrdererType: kafka
    Addresses:
        - orderer0.poc.com:7050
        - orderer1.poc.com:7050
        - orderer2.poc.com:7050
    Kafka:
        Brokers:
            - kafka0:9092
            - kafka1:9092
            - kafka2:9092
            - kafka3:9092
    BatchTimeout: 2s
    BatchSize:
        MaxMessageCount: 10
        AbsoluteMaxBytes: 99 MB
        PreferredMaxBytes: 512 KB
    Organizations:
    Policies:
        Readers:
            Type: ImplicitMeta
            Rule: "ANY Readers"
        Writers:
            Type: ImplicitMeta
            Rule: "ANY Writers"
        Admins:
            Type: ImplicitMeta
            Rule: "MAJORITY Admins"
        BlockValidation:
            Type: ImplicitMeta
            Rule: "ANY Writers"
```

Listing 5: Configuring the ordering service

In 5, we see the `Orderer` block, which specifies various aspects of the ordering service. The first few configurations state the consensus type and the addresses of orderers and brokers. The next four parameters in combination regulate how the orderers handle the distribution of transactions across batches:

- `BatchTimeout` - specifies how long the orderers wait before producing a batch of transactions
- `BatchSize`
  - `MaxMessageCount` - a batch will be split if it contains more than this number of messages
  - `AbsoluteMaxBytes` - a single batch can never occupy more space than this
  - `PreferredMaxBytes` - a batch will (preferrably) be split if it is larger than this number of bytes. However, if a transaction is larger that that, it will result in a larger batch (though never larger than AbsoluteMaxBytes)

```
Profiles:
    genesis:
        <<: *ChannelDefaults
        Capabilities:
            <<: *ChannelCapabilities
        Orderer:
            <<: *OrdererDefaults
            Organizations:
            - *Orderers
            Capabilities:
                <<: *OrdererCapabilities
        Application:
            <<: *ApplicationDefaults
            Organizations:
            - <<: *Orderers
        Consortiums:
            Banks:
                Organizations:
                - *B1
                - *B2
                - *B3

    channel-b1:
        Consortium: Banks
        Application:
            <<: *ApplicationDefaults
            Organizations:
                - *B1
            Capabilities:
                <<: *ApplicationCapabilities
```

Listing 6: Pre-generated block profiles

As stated previously, in `Profiles` we define profiles for the creation of the genesis block and the channels. In example 6, the first node defines the profile "genesis" that we will use to produce the genesis block. The last field, `Consortiums`, specifies consortiums that compose the network i.e. the categories of organizations. While every channel must belong to one consortium, a consortium can be matched to multiple channels. The second node defines the profile used to create the channel "channel-b1" and specifies (`Consortium`) both what consortium it belongs to and the organizations (exclusively B1, here) within that consortium that have access to the channel (`Application.Organizations`).

To generate the artifacts for the channel "channel-b1", the following commands 7 must be executed:

```
# to create the global genesis block for the orderers
configtxgen -profile genesis -channelID orderers -outputBlock ./genesis.block

# create the first transaction for channel-b1
configtxgen -profile channel-b1 -outputCreateChannelTx ./generate.tx -channelID
  channel-b1

# configure the channel's anchor peers
configtxgen -profile channel-b1 -outputAnchorPeersUpdate ./anchor_peers_b1.tx
 -channelID channel-b1 -asOrg B1
```

Listing 7: Generating channel artifacts

3.3.3 Deployment of Docker Containers

Since blockchain has a peer-to-peer architecture, the deployment of a network effectively corresponds to the deployment of its constituent nodes. The software for the various entities on a Hyperledger Fabric blockchain is published in the form of Docker images which must be extended, parameterized and instantiated to bring up the network. Moreover, the containers are connected by a virtual network which can also be extensively customized. This subsection deals with the configuration and deployment of these services.

Docker Compose is used to deploy the network from a definition in one or more YAML files. Although not necessary, using this tool is advised because it not only streamlines the deployment process but it also offers high-level capabilities like the inheritance of service properties and variable substitution.

```
peer:
    image: hyperledger/fabric-peer:$IMAGE_TAG
    environment:
      - CORE_VM_ENDPOINT=unix:///host/var/run/docker.sock
      - CORE_VM_DOCKER_HOSTCONFIG_NETWORKMODE=${COMPOSE_PROJECT_NAME}_poc
      - CORE_PEER_GOSSIP_USELEADERELECTION=true
      - CORE_PEER_CHAINCODELISTENADDRESS=0.0.0.0:7052
      - CORE_PEER_GOSSIP_ORGLEADER=false
      - CORE_PEER_PROFILE_ENABLED=true
      - CORE_PEER_TLS_ENABLED=true
      - CORE_PEER_TLS_CERT_FILE=/etc/hyperledger/fabric/tls/server.crt
      - CORE_PEER_TLS_KEY_FILE=/etc/hyperledger/fabric/tls/server.key
      - CORE_PEER_TLS_ROOTCERT_FILE=/etc/hyperledger/fabric/tls/ca.crt
      - CORE_VM_DOCKER_ATTACHSTDOUT=true
      - CORE_LEDGER_STATE_STATEDATABASE=CouchDB
      - CORE_LEDGER_STATE_COUCHDBCONFIG_USERNAME=
      - CORE_LEDGER_STATE_COUCHDBCONFIG_PASSWORD=
    working_dir: /opt/gopath/src/github.com/hyperledger/fabric/peer
    command: peer node start
    volumes:
      - /var/run/ : /host/var/run/
    networks:
      - poc

peer0.b1.poc.com:
    container_name: peer0.b1.poc.com
    extends:
      file: docker-compose-base.yaml
      service: peer
    environment:
      - CORE_PEER_ID=peer0.b1.poc.com
      - CORE_PEER_ADDRESS=peer0.b1.poc.com:7051
      - CORE_PEER_GOSSIP_BOOTSTRAP=peer1.b1.poc.com:7051
      - CORE_PEER_GOSSIP_EXTERNALENDPOINT=peer0.b1.poc.com:7051
      - CORE_PEER_LOCALMSPID=b1-msp
      - CORE_LEDGER_STATE_COUCHDBCONFIG_COUCHDBADDRESS=couchdb-peer0.b1:5984
    depends_on:
      - couchdb-peer0.b1
    volumes:
      - crypto-config/peerOrganizations/b1.poc.com/peers /peer0.b1.poc.com/msp :
        /etc/hyperledger/fabric/msp
      - crypto-config/peerOrganizations/b1.poc.com/peers /peer0.b1.poc.com/tls :
        /etc/hyperledger/fabric/tls
    ports:
      - 7051:7051
      - 7053:7053
```

Listing 8: Settings for a peer Docker container

Listing 8 illustrates the configuration of the Docker container for peer0 of organization B1, which uses the aforementioned inheritance paradigm by extending the service "peer". The `environment` block specifies Fabric-specific environment variables, such as TLS settings and addresses for communication between nodes. It also includes "volumes", where the location of the peer's cryptographic material is indicated; and "ports", which declares the peer's exposed ports. Every entity on the network must be configured accordingly using a similar definition; for example, the orderers' configuration must specify the path to the genesis block generated earlier.

3.3.4 Channel Creation

Once the blockchain is assembled, the next step is to deploy and set up the channels using scripts inside a special command-line interface (CLI) container intended for manual administrator functions during bootstrap. For each step, the CLI must be set to target a certain peer (which must have the necessary permissions) via exported environment variables like the peer's address and certificate information. Additionally, some calls require the CLI to also point to an orderer in the network.

The first step to initialize a channel involves the channel generation transaction that is created in 3.3.2. This transaction is submitted to the orderer, which in turn returns the channel's first block after consensus from the ordering service. Next, each peer is joined to the channel by assimilating the block into its ledger. Finally, the channel is updated regarding each organization's anchor peers. This process is displayed in 9, where we create the channel "channel-b1".

```
export ORDERER="orderer0"
export CHANNEL="channel-b1"

# a function "selectPeer" was defined which exports the specified peer's name,
  org and path to cryptographic material
selectPeer peer0 b1

peer channel create -o $ORDERER.poc.com:7050 -c $CHANNEL -f
  ./channel-artifacts/$CHANNEL/generate.tx --tls --cafile $ORDERER_CA

# must be repeated for every other peer that belongs to the channel
peer channel join -b $CHANNEL.block

peer channel update -o $ORDERER.poc.com:7050 -c $CHANNEL -f
  ./channel-artifacts/$CHANNEL/anchor_peers_$ORG.tx --tls true --cafile
  $ORDERER_CA
```

Listing 9: Initializing a channel

3.3.5 Chaincode Installation

The installation of chaincode on the channels is similarly performed by a script run inside the CLI, as is demonstrated by the installation of chaincode "smartContracts" displayed in 10. Fabric discerns two phases of the deployment of chaincode to a channel. First, the chaincode must be installed on every peer that will execute it; then, the chaincode's instantiation is a one-time action that deploys the chaincode on the channel using the specified endorsement policy and creates a corresponding Docker container on every peer.

```
export CHANNEL="channel-b1"
export CHAINCODE="smartContracts"
export CHAINCODE_VERSION="1"

# for every peer in "channel-b1", export environment variables and install
  chaincode
for i in ${B1[@]}
do
    selectPeer $(echo $i | tr "@" " ")
    peer chaincode install -n $CHAINCODE -v $CHAINCODE_VERSION -l node -p /opt/-
      gopath/src/github.com/hyperledger/fabric/peer/chaincode/smartContracts
done

selectPeer "peer0" "b1"
peer chaincode instantiate -o $ORDERER.poc.com:7050 --tls --cafile $ORDERER_CA
 -C $CHANNEL -n $CHAINCODE -v $CHAINCODE_VERSION -l node -c
  '{"Args":["com.poc.smartContracts:instantiate"]}'
```

Listing 10: Installing a chaincode on a channel

3.3.6 Distributed Network

Whereas a local network can be assembled solely using the virtual network Docker creates, distributing a blockchain across different physical hosts requires a more sophisticated setup. For this purpose, we used Docker Swarm, which allows the orchestration of a cluster (a swarm) of containers in a distributed setup.

The most significant difference between the deployment of a local network and a distributed one lies in the configuration of the Docker containers, which must include some additional fields. Under each container node, a new `deploy` section (as shown in) declares deployment configurations like replication count and node crash policies. Most importantly, it defines the hostname of the machine inside of which the container is placed.

```
deploy:
    replicas: 1
    restart_policy:
        condition: on-failure
        delay: 5s
        max_attempts: 3
    placement:
        constraints:
            - node.hostname == node_1
```

Listing 11: Additional settings for distributed networks

A Docker Swarm host can be a manager, a worker, or both. To deploy an application to a swarm, a manager node submits a service definition that specifies which container image to use and which commands to execute inside running containers. The manager node dispatches units of work called tasks to worker nodes that execute them. By default, manager nodes also run services as worker nodes.

Docker Swarm works by creating an overlay network that connects the various hosts together and serves as a bridge through which the Docker virtual network can be established. This strategy removes the need for OS-level routing between containers.

As the first step, the manager node, which in our case happens to be the B1 node, creates the overlay network and the swarm, after which a join token is emitted. For a host to join the swarm, an administrator must access the node's command line and manually submit the join token. The final step is deploying the containers as a stack of swarm nodes, rather than using Docker Compose. The whole process is shown in listing 12.

```
docker network create --driver overlay --attachable safehub
docker swarm init
docker swarm join
docker stack deploy -c docker-compose-peers.yaml safehub_peers
```

Listing 12: Deploying a distributed Fabric network

3.4 Chaincode

Hyperledger Fabric defines chaincode as a business logic package that contains one or more smart contracts. A chaincode is instantiated as a separate Docker container inside each peer that executes it on a channel and, depending on the implementation language, contains either a Go, Java or Node.js runtime. Since this proof-of-concept's business logic is implemented using the provided Node.js SDK, this section examines the former. The choice of Node.js was largely dictated by accessibility, as both the documentation for the Java SDK and the community surrounding the development of Go chaincode are comparatively sparse.

State created by a chaincode is scoped exclusively to that chaincode using a separate world state database, and can't be accessed directly by another chaincode. However, within the same network, given the appropriate permissions a chaincode may invoke another chaincode to access the latter's state.

The state of a chaincode is stored on the ledger of every peer that composes the channel that chaincode is instantiated on. However, the chaincode itself need not be installed (and thus executed) on every peer of that channel. The more peers redundantly install and run the chaincode, the more fault tolerant the chaincode becomes at the expense of efficiency. In any case, the combination of working peers must at least satisfy the endorsement policy.

This section explores the SmartContracts chaincode that implements the use case, along with the reasoning behind it.

3.4.1 Auxiliary Classes

The basis for the source code consisted of an example Node.js chaincode included in the Hyperledger Fabric sample projects repository, which relies on two key classes imported from the `fabric-contract-api` package:

`Context`
: This class provides an interface to the underlying ledger context. Most importantly, it supplies the `getState`, `putState` and `deleteState` methods through which one can access and modify the key-value records stored on the blockchain. Additionally, the proof-of-concept employs the function `createCompositeKey` which compounds an object's base key with a given namespace.

`Contract`
: Every invocable smart contract inside a chaincode is defined as an API that extends this class. It mostly abstracts internal wiring behind transaction invocations, though it does supply some useful functions. This project solely overrides the `createContext` method, which creates a custom `Context` object - referenced throughout with the variable name "`ctx`" - before every transaction is processed. This `Context` object contains the field `smartList`, an extension of `StateList` that represents a list of smart contracts.

The aforementioned sample chaincode also provides some helpful resources, namely the auxiliary classes `State` and `StateList`. These represent, respectively, an asset on the ledger uniquely identified by a key and a list of these objects. Using the above Fabric classes, they create a logical separation between records and abstract away the low-level key-value store operations. Additionally, they also cover the processes of serialization and deserialization of objects. For instance, notice the use of the Context functions inside the `addState` function of the `State` class (13).

```
addState(state) {
        let key = this.ctx.stub.createCompositeKey(this.name,
         state.getSplitKey());
        let data = State.serialize(state);
        this.ctx.stub.putState(key, data);
    }
```

Listing 13: `addState` function inside `State`

3.4.2 Smart Contract Object

The first step towards implementing the chaincode was drafting the data model of the digital contract object, which is represented by the `Smart` class[1]that extends `State`.

A contract is uniquely identified by the number of the account it is associated with, hence the field `accountNumber` which suitably serves as the record's key. To implement the signature-gathering capabilities of the digital contract, a `Smart` object contains the field `signatures`, an array that stores signatures. Each signature consists of a simple JSON object containing the name or role of the signee as the field `signee`. Finally, the field `status` is a boolean which states whether the account is active or pending. These fields are initialized upon the construction of the object as follows (14):

```
constructor(obj, accountNumber) {
    // accountNumber is used as the record's key by the constructor inherited
      from State
    // the namespace of the class is required for serialization
    super("com.poc.smartContracts.smart", [accountNumber]);
    this.signatures = []; // created empty
    this.status = false;
    Object.assign(this, obj);
}
```

Listing 14: `Smart` object constructor

`Smart` contains a host of functions, most of which were grandfathered in from the sample chaincode and deal with minor serialization and buffering tasks. For the purposes of the use case, only two methods carry relevance: `getStatus`, which simply queries the value of `status`; and `addSignature`, which handles the signature-gathering logic and will be the focus of the next paragraph.

When `addSignature` is invoked, it receives a signature as a JSON object with the format described above. The function preliminarily filters all objects with the same `signee` value out of `signatures`, before inserting the signature into the array. Finally, the object's `status` is updated to *true* if all the contract clauses have been fulfilled. This consists of checking whether each of the three required signees - the client, the account manager and an account representative - have signed the contract. The full method logic, which uses JavaScript's convenient built-in array operations, is displayed in listing 15.

[1]Since the smart contract interface was already called "`SmartContract`", the class that represents the actual contract was simply named "`Smart`".

```
addSignature(signature) {
    this.signatures = this.signatures.filter(s => s.signee !==
     signature.signee);
    this.signatures = this.signatures.concat([signature]);

    if (
        this.signatures.some(s => s.signee === "client")
        &&
        this.signatures.some(s => s.signee === "account_manager")
        &&
        this.signatures.some(s => s.signee === "bank_representative")
    )
        this.status = true;
}

getStatus() {
    return this.status;
}
```

Listing 15: `addSignature` operation inside `Smart`

3.4.3 Business Logic Interface

Chaincodes in Hyperledger Fabric can contain an arbitrary number of smart contracts invocable by their namespace. Since this proof-of-concept's business logic is rather unelaborate in terms of the complexity of endpoints, only a single interface was necessary to manage the state of the chaincode.

This controller has the name "`SmartContract`" and contains the 4 externally visible endpoints necessary for the use case (shown in listing 16): `createSmart`, `getSmart`, `addSignature` and `deleteSmart`.

```
createSmart(ctx, data) {
    let dataParsed = JSON.parse(data);
    let accountNumber = dataParsed.accountNumber;
    let obj = {
        bank: ctx.stub.getCreator().mspid.split("-")[0],
        timestamp: new Date(Date.now()).toISOString(),
        data: dataParsed
    };
    let smart = new Smart(obj, accountNumber);

    ctx.smartList.addSmart(smart);
    return smart;

```

```
addSignature(ctx, accountNumber, signature) {
    let smartKey = Smart.makeKey([accountNumber]);
    let smart = ctx.smartList.getSmart(smartKey);
    smart.addSignature(JSON.parse(signature));
    ctx.smartList.updateSmart(smart);
    return smart.getStatus();
}

getSmart(ctx, accountNumber) {
    let smartKey = Smart.makeKey([accountNumber]);
    let smart = ctx.smartList.getSmart(smartKey);
    return smart;
}

deleteSmart(ctx, accountNumber) {
    let smartKey = Smart.makeKey([accountNumber]);
    ctx.smartList.deleteSmart(smartKey);
}
```

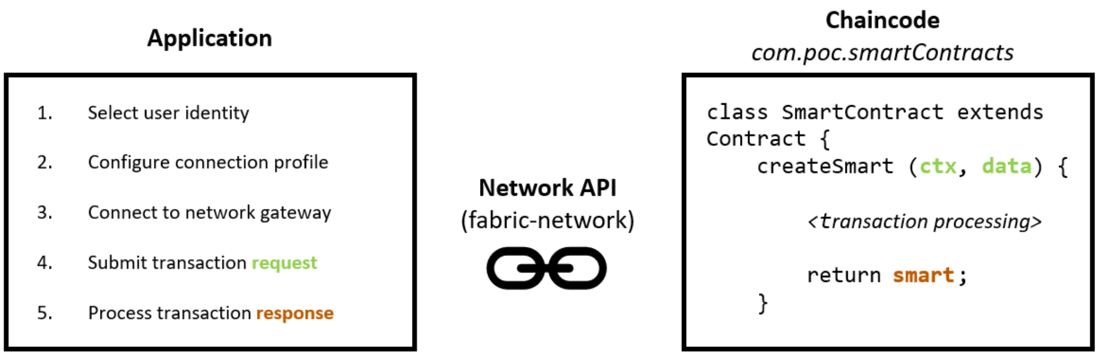
Listing 16: All invocable endpoints endpoints inside `SmartContact`

All of these functions access the ledger through `ctx.smartList`, which itself invokes the methods of the `StateList` class. For example, to add a contract to the ledger, an instance of `Smart` is created from the received JSON data and then that object is passed as input to the appropriate `addSmart` function. To consult the ledger, we prepare a composite key that is used to query the record with the corresponding key.

3.5 Application

Once a chaincode is deployed on a channel, its operations can be invoked by an external user by establishing direct connections to peers and orderers. Although these interactions can be performed with the use of command-line scripting, practical business applications call for high-level interfaces. The Hyperledger Fabric project supplies multiple Node.js modules intended for interaction with the blockchain, among which "fabric-network" is the recommended API which applications can use to invoke deployed smart contract functions.

The flow of an application submitting a transmission proposal consists of a series of steps (illustrated in figure 3.4), three of which are necessary setup configurations and the remaining two are the actual communication of JSON objects between application and chaincode. These phases deal with concepts that are the focus of the next sections.


Application
1. Select user identity
2. Configure connection profile
3. Connect to network gateway
4. Submit transaction request
5. Process transaction response
Network API
(fabric-network)
Chaincode
com.poc.smartContracts

```
class SmartContract extends
Contract {
    createSmart (ctx, data) {

        <transaction processing>

        return smart;
    }
```

Figure 3.4: A typical application flow consists of three configuration steps followed by a transmission request and response.

3.5.1 User Identity and Wallet

Hyperledger Fabric's PKI-like access control model means that chaincode invocation necessitates user authentication in the form of a digital certificate corroborated by a trusted CA. Every user possesses a set of these certificates, abstracted as a wallet, each of which representing an identity or alias of that user. Each identity has a set of assigned roles on specific channels and may be issued by a different CA. When an application interacts with the ledger, the user must specify an identity from their wallet.

To create an identity, a set of credentials (perhaps generated using the methods described in 3.3.1) must be procured. Next, using SDK operations the cryptographic material of the user is packaged into an identity, adding headers and an MSP label. Finally, the identity is injected into a pre-generated wallet. The wallet may be stored in a variety of more or less safe media; for the purposes of the proof-of-concept, a filesystem wallet is sufficient. This flow is demonstrated in listing 17.

```
// Load credentials from MSP
const credPath = path.join(CRYPTO_PATH, `/peerOrganizations/${WALLET_ORG}.poc.com/users/${WALLET_USER}@${WALLET_ORG}.poc.com`);
const cert = fs.readFileSync(path.join(credPath,
  `/msp/signcerts/${WALLET_USER}@${WALLET_ORG}.poc.com-cert.pem`)).toString();
const key = fs.readFileSync(path.join(credPath,
  `/msp/keystore/keystore_sk`)).toString();

// Generate identity from credentials
const identityLabel = `${WALLET_USER}@${WALLET_ORG}.poc.com`;
const identity = X509WalletMixin.createIdentity(`${WALLET_ORG}-msp`, cert, key);

// Create wallet and import identity
const wallet = new FileSystemWallet(path.join(IDENTITY_PATH,
  `userWallets/${WALLET_ORG}`));
wallet.import(identityLabel, identity);
```

Listing 17: Generating a wallet

3.5.2 Network Gateway and Connection Profile

After a user identity is selected, the application must connect to the network using a network gateway interface, an abstraction of the communication mechanisms required for interaction with the network. The network gateway provides a transparent separation of the connection scheme from the rest of the application processes. For example, different connection profiles can be defined for individual organizations or groups within the same org; meanwhile, the actual application code repository need not changed and can be shared between those entities.

To connect to the network gateway, the application must configure the target subsection of the network topology by loading a connection profile. This YAML specification details the interaction process and is located inside the file "networkConnection.yaml". The connection profile declares what entities - organizations, peers and orderers - are available and their respective TLS certificates, as shown in listing 18 (for the sake of simplicity, the connection profile in the example is cut down to one peer and one orderer).

```
channels:
  channel-b1:
    orderers:
        - orderer0.poc.com
        - orderer1.poc.com
        - orderer2.poc.com
    peers:
      peer0.b1.poc.com:
        endorsingPeer: true
        chaincodeQuery: true
        ledgerQuery: true
        eventSource: true

organizations:
  B1:
    mspid: b1-msp
    peers:
      - peer0.b1.poc.com
      - peer1.b1.poc.com

orderers:
  orderer0.poc.com:
    url: grpcs://localhost:7050
    tlsCACerts:
      path:
        crypto-config/ordererOrganizations/poc.com/msp/tlscacerts/tlsca.poc.com-
        cert.pem
    grpcOptions:
      ssl-target-name-override: orderer0.poc.com

peers:
  peer0.b1.poc.com:
    url: grpcs://localhost:7051
    tlsCACerts:
      path: crypto-
        config/peerOrganizations/b1.poc.com/peers/peer0.b1.poc.com/msp/tlscacerts/tlsca.b1.poc.co
        cert.pem
    grpcOptions:
      ssl-target-name-override: peer0.b1.poc.com
      request-timeout: 120001
```

Listing 18: Network profile for the application

3.5.3 Transaction Request and Response

The final step of the application-side transaction process is the handling of the transaction itself i.e. its request-response cycle. First, the application must specify the chaincode and smart contract names and retrieve an object which represents a smart contract instance on the network. Next, the request is dispatched using the `submitTransaction` function, given the target endpoint name and its arguments. Using a gateway, the developer need not concern themselves with the three steps of the transaction lifecycle: the entire process is performed automatically and transparently. Finally, as soon as a response is returned, the transaction output can be processed.

The complete application code for the `createSmart` function, which receives a JSON object representing an account opening contract and creates a corresponding instance on the blockchain, is displayed in listing 19. This sample is only part of the source code for an example client-side REST API that invokes chaincode operations, which should include a similar module for each such operation.

```
function main(data /* account object */) {
    // Load connection profile
    let connectionProfile = yaml.safeLoad(
      fs.readFileSync(
        path.join(NETWORK_CONNECTION),
        "utf8"
      )
    );

    // Set connection parameters
    let connectionOptions = {
      identity: `${WALLET_USER}@${WALLET_ORG}.poc.com`;,
      wallet: wallet,
      discovery: {
        enabled: false,
        asLocalhost: true
      }
    };

    // Access network
    const gateway = new Gateway();
    gateway.connect(connectionProfile, connectionOptions);
    const network = gateway.getNetwork(`${CHANNEL}`);

    // Access smart contract
    const contract = network.getContract(
      "smartContracts",
      "com.poc.smartContracts"
    );

    // Submit transaction request and receive response object
    const response = contract.submitTransaction(
        "createSmart",
        `${JSON.stringify(data)}`
    );
    let smart = JSON.parse(response.toString());

    // Signal the garbage collection of the gateway
    gateway.disconnect();
}
```

Listing 19: Full code of the application-side `createSmart` operation

3.6 Transaction Lifecycle

As stated before, Hyperledger Fabric features a pluggable consensus model, which means that the consensus algorithm can be customized per channel from a collection of possible modules. Regardless of the algorithm chosen, the packaging of transactions into blocks is performed by a cluster of orderer nodes. However, consensus is only one of the three phases that comprise a transaction's lifecycle.

1. The application proposes a transaction to some endorser peers. Endorsers execute containerised chaincode and return a proposal response (containing their endorsement in the form of a digital signature) to the application without (yet) applying the state deltas to their copy of the ledger. A pre-determined minimum combination of endorsers must endorse the transaction, a condition that will be checked later.

2. The application distributes the transaction proposal which contains endorsements to the orderers. The ordering service executes the consensus mechanisms in order to produce a total order of transactions bundled into blocks. However, orderers do not perform transaction validity checks: they merely produce a consistent and final chain of blocks. Orderers append the created block to their personal copy of the ledger and propagate it to every peer in the channel.

3. Every peer independently evaluates the validity of the transactions contained in the incoming block. Next, the block is appended to the chain and the world state is changed accordingly. This includes checking the compatibility of transactions with the previous state of the ledger, the presence of the necessary endorsements and the hash of the block. Finally, the client application is notified about the ordering, validation and commitment of the transaction.

It should be noted that, because the ordering service produces an immutable and final ledger, peers do not reject blocks but rather mark invalid transactions as such. This way, all chaincode invocations, even invalid or idempotent ones, are recorded on the blockchain.

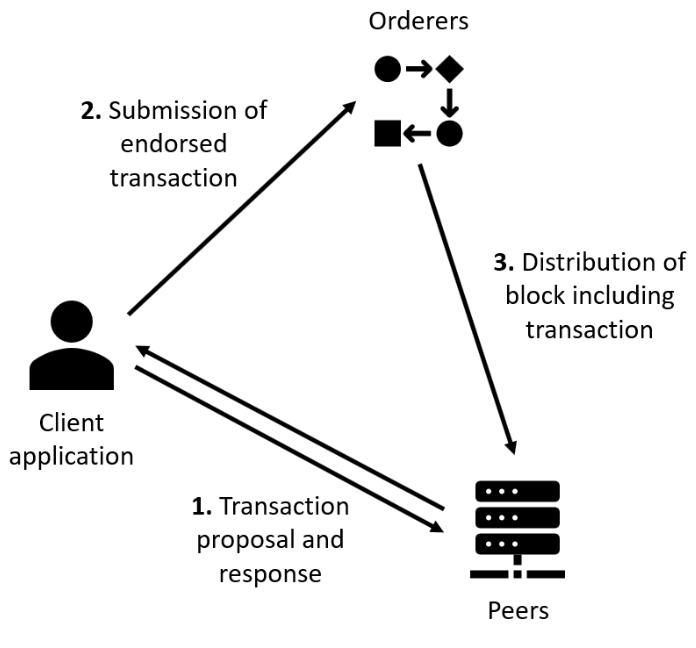

Orderers
2. Submission of endorsed transaction
3. Distribution of block including transaction
Client application
1. Transaction proposal and response
Peers

Figure 3.5: During its lifecycle, a transaction is exchanged between the application, endorser peers and the ordering service.

Chapter 4

Integration with Onboarding

We have examined the creation of a smart contracts service running inside a blockchain, which when applied to the account opening process would act as a tool to better define the relationship with the client. Yet, we haven't discussed the role of the smart contract and the encompassing blockchain inside a bank's infrastructure. This chapter presents the surrounding architecture and how blockchain was integrated into it in order to create a complete proof-of-concept. First, however, a brief introduction of the bank infrastructure involved in the account opening process is needed.

4.1 Bank Infrastructure

The most integral part of every bank's architecture is the banking core, the back end system that processes and supports vital functions across all the bank's branches. The core is built around the bank's internal database, a distributed data store that stores all the bank's data, and is physically and logically separated from all non-essential systems and services.

Banks establish an infrastructure around the banking core, consisting of multiple channels (like ATMs, call centers and e-mail services) via which customers can access services. However, in the last decades, the advent of the Internet and the demands of an increasingly digital society have translated into a push for online banking.

Modern banks incorporate a so-called digital channel i.e. the platform containing all the services and solutions that compose online banking, most of which have been moved from the traditional branch banking experience to a mobile approach. It is also this channel that the blockchain network and the smart contracts contained therein integrate[1].

[1]Blockchain is not limited to online solutions; if employed strictly as an immutable form of data store, blockchain could in theory also implement the database model of the banking core.

4.1.1 Onboarding

In a banking context, the account opening process is referred to as onboarding and is initiated by either a client or a bank branch personnel member operating on an onboarding application. The onboarding application mediates the process by communicating with other bank APIs within and beyond the digital channel. Namely, upon opening and closing of onboarding, it transmits the details of the process to the core layer of the bank, which stores the customer and account data on the internal database.

4.2 Blockchain Proof-of-Concept

In a blockchain-enhanced architecture, the behaviour of both the pre-existing onboarding application and banking core is not replaced but rather the actions outlined above are supplemented by interactions with a blockchain.

In order to demo a complete user story, it is necessary to assemble a dummy bank framework onto which to nest a blockchain system. The full proof-of-concept architecture that was devised consists of the following components (as is also illustrated in figure 4.1):

- a Hyperledger Fabric blockchain consisting of a network of nodes that execute the chaincode SmartContracts, perform a consensus algorithm and replicate the ledger
- an API that abstracts away the underlying blockchain business logic by externalizing operations as REST endpoints
- a simulated banking core consisting of a database which stores client and account data, and is accessible via a second API
- an onboarding front end, which serves as the end-user interface and communicates with both the blockchain and banking core APIs

Since the blockchain side of the proof-of-concept has already been thoroughly examined in the last chapter, we will now discuss the three remaining elements of the proof-of-concept.

4.2.1 Blockchain API

As stated in 3.5, the blockchain interfaces with applications that use the fabric-network module. Such an API was developed in this project for testing purposes using the procedures outlined in that section and illustrated by the code listings contained therein, and nested inside a REST API.

To restate, the `SmartContract` controller implemented in the blockchain's chaincode back-end declares the following 4 methods: `createSmart`, `getSmart`, `addSignature` and `deleteSmart`. As the link between chaincode and other components, the blockchain API

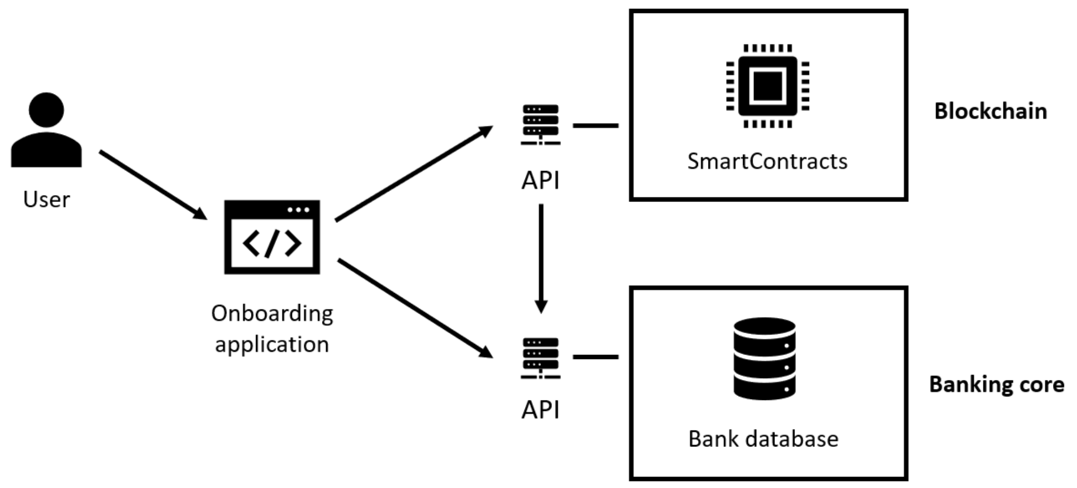


Figure 4.1: The place blockchain occupies inside the onboarding architecture and the interaction links between components.

must reciprocate these functions in the form of outward-facing REST endpoints, resulting in the following set of available REST operations:

- `POST smartContracts`
  creates a smart contract object on the blockchain, given a JSON object containing the account number
- `GET smartContracts/{accountNumber}`
  retrieves smart contract object from the blockchain that corresponds to the given accountNumber
- `POST smartContracts/{accountNumber/}signatures`
  seeks the corresponding smart contract object and adds the given signature object to it (which must contain the name/role of the signee)
- `DELETE smartContracts/{accountNumber}`
  deletes the smart contract object identified by the given accountNumber, which corresponds to a contract cancellation

The blockchain API was implemented using Loopback, a Node.js framework that allows the creation of JavaScript APIs. A REST controller maps each endpoint to a corresponding Node.js module, which in turn invokes the appropriate SmartContract chaincode operation.

Listing 20, shows a snippet of the `SmartContracts` controller (not to be confused with the similarly named chaincode interface) source code. Here, the REST endpoint `createSmart` is declared and the invocation of the corresponding `createSmart` application module is assigned to it (the latter of which's source code was displayed in 19).

```
SmartContracts.createSmart = function (accountNumber, cb) {
    // invoke the createSmart application module
    createSmart.main(accountNumber)
        .then(result => cb(null, result))
        .catch((e) => {
            // error handling
            var err = new Error(e);
            err.statusCode = 500;
            err.message = e;
            return cb(err);
        });
};

// declare REST endpoint
SmartContracts.remoteMethod('createSmart', {
    description: 'Creates a new smart contract.',
    http: {
        path: '/',
        verb: 'post',
        status: 200,
        errorStatus: 400
    },
    accepts: {
        arg: 'data',
        type: 'object',
        http: {
            source: 'body'
        }
    },
});
```

Listing 20: API REST controller

4.2.2 Onboarding Front End

As part of the business setting inside Novabase, this project was given access to a development version of Wizzio, one of Novabase's financial services solutions. Wizzio is a platform which allows the implementation of a wide variety of business use cases as "journeys", highly-flexible user stories that are well-defined by customizable constraints. Journeys are assembled using a dedicated journey designer and exported as stand-alone applications. Internally, Wizzio is built on a Java microservices architecture using Apache Karaf.

The donated version of Wizzio features an onboarding application simply codenamed "Onboarding". In addition to implementing a bank account opening journey, Onboarding features a functional and aesthetic JavaScript front end using the React framework, which approximates a realistic end-user experience. Thanks to this contribution, we were able to streamline the development process, since the front end source code was merely tweaked to communicate onboarding requests to the newly integrated blockchain API in lieu of the existing microservice API.

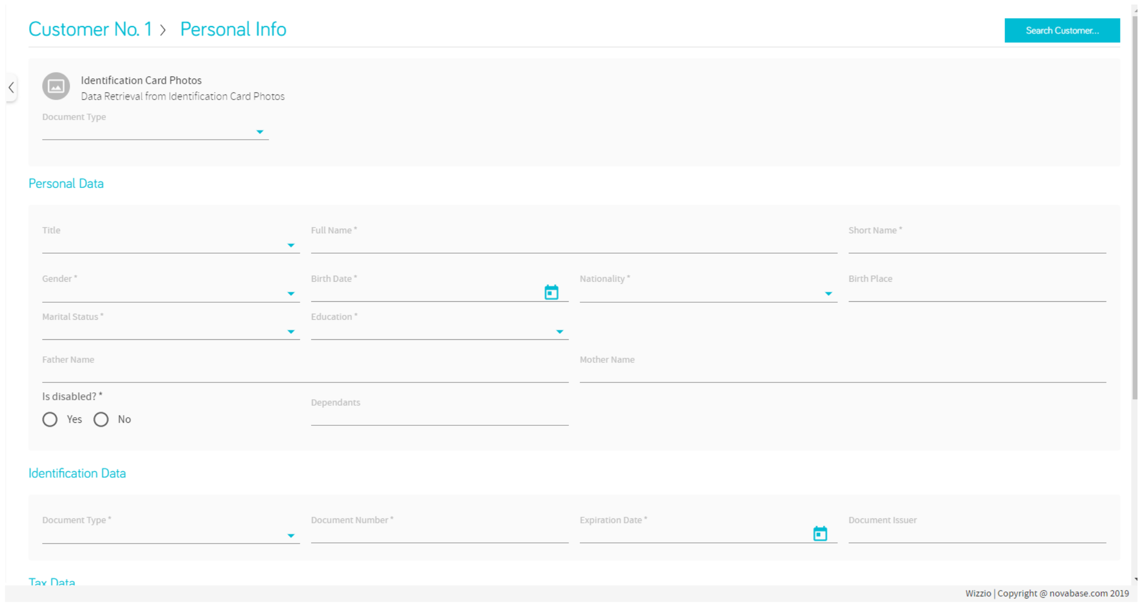


Figure 4.2: The look of the onboarding front end is similar to what is expected in a production-ready setting.

The onboarding application guides the user through a series of forms that must be filled with details regarding the account, such as the user's name and residency. Additionally, some digital records must be uploaded, including a photograph of the client's identification document. Provided all form fields are filled appropriately, the application submits the onboarding request to the blockchain API and the journey advances to the signature-gathering phase. The user is presented with a widget that lets them provide their signature in a digital form, which is then similarly transmitted to the blockchain API. All exchanges between the application and the blockchain API are performed as calls to REST endpoints, as is showcased in listing 21.

```
// to create a smart contract
let config = {
    data: contractData,
    url: `http://localhost:4005/api/smartContracts`,
    method: "post"
};

// to submit a signature
```

```
let config = {
    data: signatureData,
    url: `http://localhost:4005/api/smartContracts/${accountNumber}/signatures`,
    method: "post"
};

const result = await httpClient.request(config);
```

Listing 21: Onboarding application connection to the blockchain API

4.2.3 Banking Core

Finally, a working banking core was simulated by constructing a small database whose schema is captured in 22. The database mainly consists of two tables, `client` and `account`, which hold client and account data respectively. Although not essential to the smart contract use case, these tables were expanded with additional fields (such as `nationality`) to confer authenticity to the proof-of-concept. A third table, `client_account`, serves as a joining table to allow for N:N relationships between the previous two. This banking core database was coupled with a corresponding Loopback REST API, to abstract the underlying DBMS and provide a standard HTTP interface composed of externally accessible operations.

```
create table client
(
    nif int unsigned not null,
    fullName varchar(255),
    idDocNumber int unsigned,
    birthDate date,
    nationality varchar(255),
    email varchar(255),
    primary key (nif)
);

create table account
(
    number int unsigned not null,
    status boolean not null default 0,
    primary key (number)
);

create table client_account
(
    customerNif int unsigned not null,
    accountNumber int unsigned not null,
    foreign key (customerNif) references client(nif),
    foreign key (accountNumber) references account(number)
);
```

Listing 22: Banking core database schema

4.2.4 Flow of Information

Upon receiving user input, the onboarding application submits a request to the blockchain API, which spawns and stores on the ledger a smart contract object representative of the process. From here, the signature-gathering phase is governed by the onboarding application, which continues to contact the blockchain via REST requests to the blockchain API. Every time the API submits a signature to the chaincode, it evaluates the returned status of the contract. When this result signals that all contract clauses are fulfilled, the API directly notifies the banking core about the activation of the account.

This flow of information can also be conceptualized as the following list of steps:

- The customer initiates an account request:

  1. the client inputs the account opening details into the Onboarding application
  2. if valid, Onboarding transmits the account request to the blockchain API via the `POST smartContracts` endpoint
  3. the API connects to the blockchain and invokes the `createSmart` chaincode method, creating a smart contract object representative of onboarding

- After the smart contract object is created, its mechanisms can be activated:

  4. each signee - customer, account manager or bank representative - confirms its approval of the contract
  5. Onboarding transmits each individual signature to the blockchain API via the `POST smartContracts/{accountNumber}/signatures` endpoint
  6. the application once again connects to the blockchain invokes the chaincode, this time signing the smart contract object with the `addSignature` operation

- Finally, once all necessary signatures are gathered, the contract takes effect:

  7. the application identifies the smart contract's activation by the last signature submission and propagates this event to the banking core API

Even if the contract is aborted half-way through the process, all interactions - creation, signing, deletion - are stored permanently on the ledger. Moreover, at any step of the process, the smart contract state can be queried about its status or the presence of signatures.

Chapter 5

Performance Testing & Evaluation

While previous chapters described the blockchain network and the surrounding proof-of-concept infrastructure, this chapter handles its evaluation from a performance point of view. Detailed here is the design of a test battery and what considerations were take, the performance environment set-up, and the actual evaluation of the system. The latter comprises a discussion of the test results and their contextualizing to conclude about the proof-of-concept's viability in integrating an existing bank's Onboarding architecture.

5.1 Test Design

5.1.1 Preliminary Considerations

Initial test designs revealed that the performance of our infrastructure is critically limited by the number of requests it can handle concurrently. Upon closely examining the API logs, it was concluded that network saturation can cause RPC communication with peers/orderers to fail. If a request fails, the SDK will perform a number of retries. However, especially heavy congestion can cause all retries for a particular transaction to fail, resulting in the loss of the transaction. For instance, submitting 200 simultaneous requests to the API may yield only 190 successful transactions - the remaining 10 will fail to be registered on the blockchain and return an error. As the parallel execution demands of the network grow, a feedback loop emerges where congestion causes requests to fail and subsequently linger for longer periods, which sustains the aforementioned congestion.

It was assessed, once more by inspecting the application logs, that the actual execution of the chaincode was a weak contributor to the transaction times. In fact, the most time was taken up by network tasks like propagating events and reaching consensus.

It was therefore decided that the most adequate evaluation strategy would be to test the concurrent computing capabilities of our network. In other words, we want to determine how many simultaneous transactions the blockchain infrastructure can handle and how increasing concurrency demands affect the quality of service. From a point of view anchored in common sense and general observations of a host of other multi-threaded distributed systems, it is reasonable to hypothesize that, as computational and networking demands become more strenuous, the system will suffer a gradual decrease in throughout.

5.1.2 Test Implementation

A test battery was devised that is divided into rounds. In each round, we execute a number of simultaneous requests to the API endpoint `POST smartContracts`, which then calls the chaincode-side `createSmart` operation. At the end of each round, the outcome of the test is collected into a results file, which will be analysed in spreadsheet form at a later stage to draw conclusions. Specifically, we will be recording the total time elapsed and the number of transactions that returned a success status.

As mentioned earlier, transaction times are highly dependent upon the surrounding network processes which are managed by the application. Due to the way the application and the blockchain are intertwined, it is counter-productive to examine the performance of the blockchain in isolation. However, a second batch of test rounds was performed where the API code was tweaked to bypass the request to the chaincode, leaving only the other fabric-network tasks described in 3.5. This allows us to scrutinize the performance of the application in separate, which might reveal some insight into the network's processing bottlenecks.

5.1.2.1 Scripts

The test is implemented as a combination of two scripts: a Node.js script implements the actual test round, while a Bash script invokes the former multiple times as a sequence of rounds using different parameters.

The Node.js script (code is shown in 23) submits a parameterizable number of parallel requests to the blockchain API using the "request-promise"library, which bundles the requests into a single `Promise` object. The script holds execution until every request has produced a status response, in effect waiting for every request to be fulfilled or to fail. Next, it examines the response of every request and aggregates the number of successes and the total execution time into a results digest. This compiled result is then appended into a common results file.

```
let main = function (N_REQUESTS) {
    for (var i = 0; i < N_REQUESTS; i++) {
            calls.push(
                request(
                    {
                        url: `http://localhost:3006/api/smartContracts`,
                        method: "POST",
                        headers: {
                            'Accept': 'application/json',
                            'Content-Type': 'application/json'
                        },
                        body: JSON.stringify(
                            {
                                accountNumber: accountNumber
                            }
                        )
                    }
                ).catch(err => function (err) { })
            );
    }

    (async () => {
        let start = Date.now();
        const results = await Promise.all(calls);
        let end = Date.now();
        let succ = results.filter(y => y === '').length;
        let result = `${N_REQUESTS}\t${succ}\t${end - start}\n`;
        fs.appendFileSync('test-results.csv', result);
    })();
}
main(process.argv[2]);
```

Listing 23: Test round implementation

On the other hand, the Bash script (shown in listing 24) merely executes the instruction that invokes the Node.js script. This command is placed inside the innermost of two nested for loops. The first controls the N_REQUESTS parameter, which states how many concurrent requests to submit in one test round. The second merely invokes j rounds for every value of the N_REQUESTS parameter, where the value of 10 was chosen for j. The results for every value of N_REQUESTS will later be averaged to yield a consistent performance curve.

```
# values of N_REQUESTS for full-system rounds
for i in 10 20 40 60 80 100 120 140 160 180 200 220 240 260 280 300 320 340 360
  400 450 500

# values of N_REQUESTS for application-only testing
for i in 10 20 50 100 200 300 400 500 600 650 700 750 800 850 875 900 925 950

```

```
do
for j in 1 2 3 4 5 6 7 8 9 10
do
    node ./test.js $i
done
done
```

Listing 24: Bash script that invokes the Node.js test round

The range of values (also present in 24) for `N_REQUESTS` was picked based on an early observation of test results for a variety of experimental values that was conducted to understand what a meaningful range of values should look like.

For the full-system test battery, a gradual increase in increments of 20 transactions was decided upon. In addition, some higher values were also sampled to assess how well the system handles heavier loads. In regards to the application-only system, its performance was empirically gauged as much higher, while a steep decline was noticed around the 750 transactions mark. Accordingly, higher values of `N_REQUESTS` were inspected much more granularly, while the overall range of values was magnified in comparison.

5.2 Setup

Unfortunately, a proper benchmarking environment could not be mustered due to a lack of resources. Due to being a proof-of-concept, the allocation of the necessary resources to the project could not be justified from the financial point of view of Novabase. Instead, performance tests were performed on a single personal machine running a Linux distribution natively with the specifications presented below. In regards to software, all required dependencies and their versions are as outlined previously in 3.2.

CPU i5-8250U @ 1.60 GHz (4 physical cores)

RAM 8 GB

Hard drive 128 GB, HDD

Operating System Ubuntu 18.04 LTS, 64-bit

The tests targeted the local version of the network, which consists of the following Docker containers all under the same localhost network:

- 6 peers
- 3 orderers
- 4 Kafka brokers
- 3 Zookeeper nodes [1]
- 6 CouchDB peer databases
- 1 CLI administrator

Moving into the results section, please bear in mind the constraints these conditions place on the performance test results. First of all, the processing speed and memory capacity are far from what is expected from a proper benchmarking environment. Secondly, such a high number of memory-containers which regularly and concurrently perform RPC communications with other components places significant stress on the network, given its local setup. Furthermore, chaincode containers, which are not accounted for in our description of the network topology, occupy yet more space in memory.

5.3 Evaluation

5.3.1 Test Results

After the execution of the tests, the resulting data was collated from the output file and manipulated in Microsoft Excel to produce the following plots (5.1 & 5.2). Each graph combines the curves of both the parameters that were measured in the tests performed. The blue line represents the relationship between number of tra nsactions and success rate, while the orange line represents the relationship between the number of transactions and total elapsed test time. On the bottom, the legend displays the numbers of transactions used in the tests, while the left and right-hand side axes show reference values for the line of the same colour.

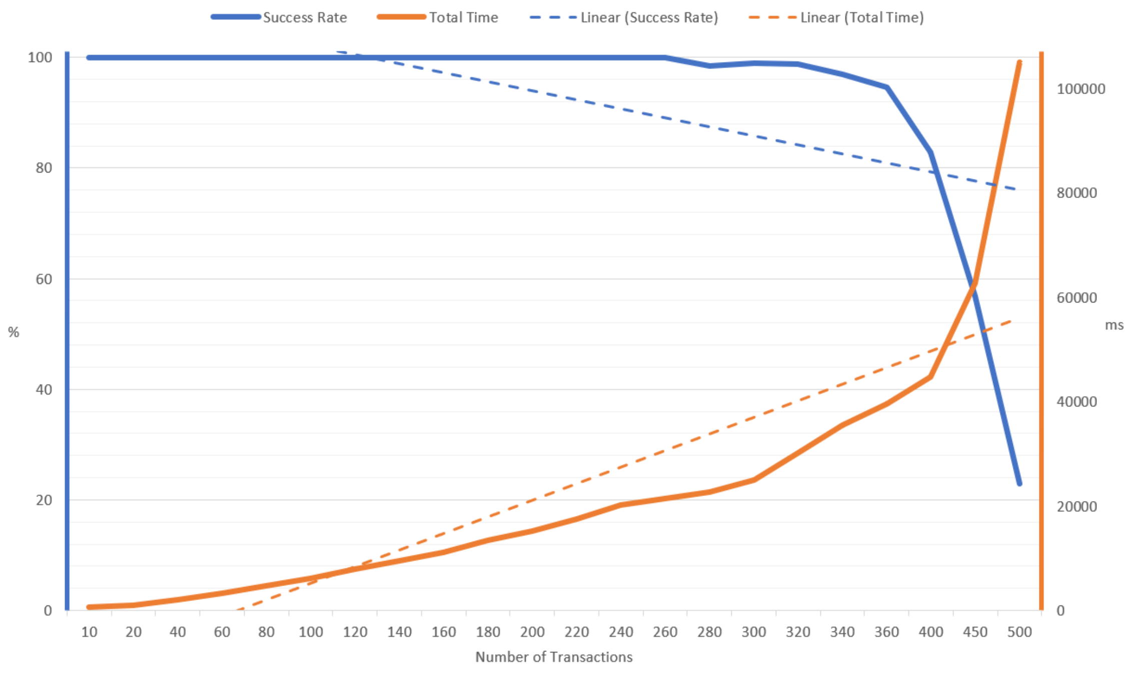


Figure 5.1: Total time and success rate results of the full-system performance tests

[1]Although not discussed in the previous sections, Apache Zookeeper is a vital component of the inner workings of a Kafka-based consensus system in Hyperledger Fabric. For more information, please refer to the Fabric documentation.

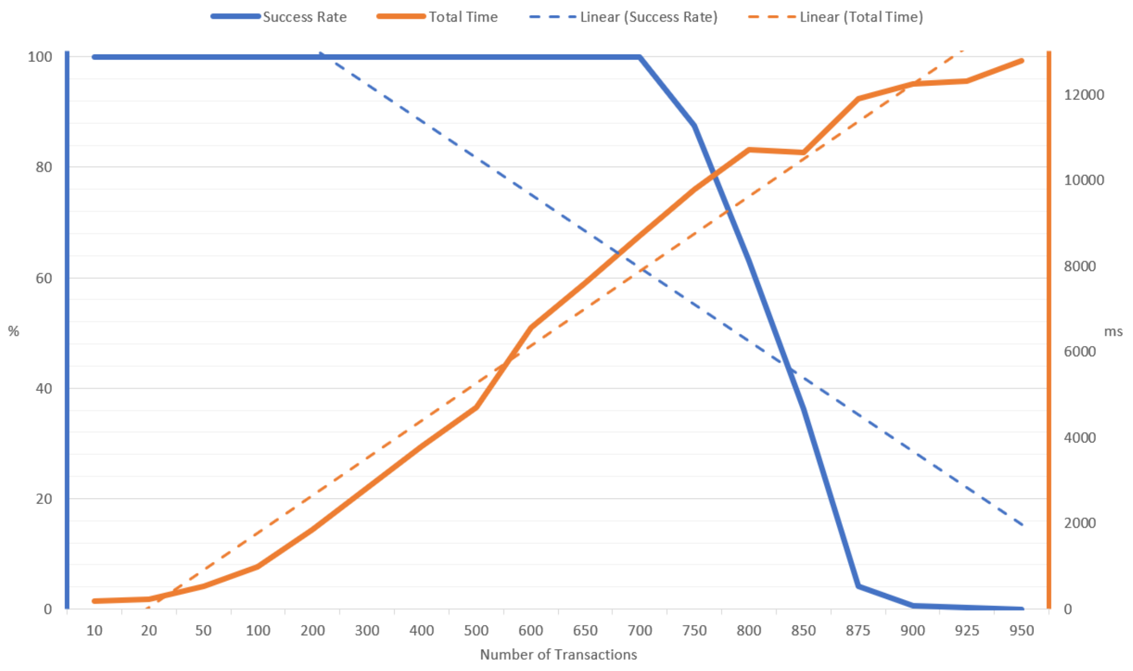


Figure 5.2: Total time and success rate results of the application-only performance tests

5.3.2 Interpretation of the Test Results

As was hypothesized, the biggest contributing factor to the performance of the blockchain system is its ability to handle strenuous numbers of parallel requests. We can surmise from contemplating the test results charts that the system can handle up to 240 simultaneous transaction requests without failures, at which point the infrastructure starts to break down. At around 360 transactions, interference between requests and the resulting congestion of the network becomes rampant, at which point success rates plummet while the total elapsed time increases dramatically. TPS stays stable at around 10 TPS, reaching a peak value of 19 at 40 parallel transactions. As to why exactly transactions fail in conditions of heavy concurrent execution demands, that was not examined due in part to the limitations outlined in 5.3.4. Discerning whether computing or networking power is the underlying bottleneck would constitute a sound cause for further study.

As was also suspected, the performance in the application-only tests is much superior, reaching a peak of 106 TPS with 500 simultaneous requests. We can therefore deduce that the consensus mechanisms introduce a performance bottleneck into the system, most likely due to the added complexity of the communication of transactions in a distributed system.

5.3.3 Viability of Integration

Even though a blockchain-enhanced architecture could be envisioned to form the backbone of a bank's digital channel, the evaluation of the proof-of-concept must pertain to the specific use case of opening an account. As such, we will consider the demands of the process of Onboarding for a typical bank.

Unfortunately, such metrics are not publicly available, as they constitute confidential information deemed incredibly valuable by banks. Nonetheless, Novabase, as the provider of financial services for many important banks, has access to some estimations of the daily statistics of Onboarding. In one of its biggest clients, the number of daily bank account openings averages around 2000, yielding a TPS value of 0.02. Accounting for the irregularity of digital channel access patterns throughout the day and year, and the possibility of a client with larger demands, a reasonable TPS value would still not be greater than 1. In fact, even considering that the infrastructure of back-end processes behind opening a bank account is vastly more complex than the admittedly minimal information flow of the SmartContracts chaincode and associated application / API, an Onboarding system would not conceivably need to maintain a transaction rate of above 10 TPS (a hundredfold increase).

In such light conditions, the concerns about the added complexity of blockchain are blunted and we can, therefore, declare the technology as acceptable for the use case at hand.

5.3.4 Evaluation Applicability & Ideal Environment

The relevancy of our conclusions is challenged by the conditions in which the tests were performed. As described above, due to the lack of justifiable financial backing, the testing environment is far from adequate. A proper set-up would require a computer designed specifically for testing with, in addition to powerful primary and secondary memory capabilities, a fast processor with many cores such as to optimize for parallel execution.

Although a functioning distributed version of the blockchain network was developed for this project, the circumstances impeded its testing. As was deduced earlier, the performance of our infrastructure is limited by the number of requests the blockchain can handle concurrently before congestion of the network causes transaction proposals to fail due to connection timeout. Not only was this capacity exceedingly low in our testing environment, but it also varied wildly and arbitrarily between tests, shortchanging the reliability of the results. For these reasons, it would be recommended to redo the tests in a distributed set-up in order to both improve performance by partitioning computation and better emulate a real setting.

Finally, given the shortcomings above, the tests were never conducted with the intent of accurately portraying the proof-of-concept's performance profile. Instead, we strove to gather knowledge about the bottlenecks of the system as well as the viability of the proof-of-concept performance-wise. Proper performance tests would not be designed around bursts of transactions. Rather, they would test the system under gradually increasing TPS conditions to assess under what stress conditions a breakdown in quality-of-service occurs.

In closing, for a thorough and well-structured performance benchmarking of a Hyperledger Fabric network, please refer to the following 2018 paper[13], which tests a host of parameters in an excellent setup.

Chapter 6

Conclusions

This last chapter marks the end of the document and represents a valuable opportunity to synthesize our findings and extract new knowledge. It is time to review the stated goals, evaluate the proof-of-concept in context and contemplate the immediate and future applicability of this body of work.

6.1 Summary

This dissertation started by establishing a problem: the conventional contract is a remarkably important yet old-fashioned tool. Banks, who have long realized the traditional contract's inadequacies in terms of inefficiency and security, desire a more streamlined and dynamic solution to some of their financial services use cases. One such situation is the opening of a bank account, where the relationship with the client is ill-defined in the period between the request and creation of that account. As part of the digital revolution of the last decades, the smart contract, a blockchain-based digital form of contract, has gained relevance. After reviewing its properties and potential to remedy the pitfalls of the traditional contract, we proposed a smart contract service running on a blockchain network as a way of creating an intermediate immutable relationship between a bank and a customer in the time frame mentioned above.

Chapter 2 presented a brief overview of some crucial distributed systems and cryptography concepts. This was followed by an examination of blockchain technology - an innovating mutation of peer-to-peer networks that stores data as a long chain of cryptographically-linked blocks, guaranteeing the immutability of the data. We discussed how blockchain can be described as public or private and the implications of selecting one over the other, namely in regards to consensus algorithms. It was concluded that, due to the privacy concerns of the business setting, the use case requires a private blockchain.

As of writing, this choice limits the state-of-the-art available to the blockchain framework Hyperledger Fabric.

Chapter 3 detailed the nuances of constructing a proof-of-concept composed of a private Hyperledger Fabric network, the codebase for the business logic of the use case, and user application to invoke the chaincode operations that access the ledger.

Next, in chapter 4 we considered the place a blockchain network running a smart contracts service occupies in a typical bank's onboarding infrastructure, complete with a description of the system that was developed to assemble a complete proof-of-concept.

This infrastructure was put to the test by evaluating its performance in stress tests in chapter 5, where we drew the conclusions outlined therein. In short, we assessed a blockchain-enhanced architecture as adequate for the Onboarding use case.

In closing, we developed a software solution to a Financial Services problem based on a great deal of research into blockchain and smart contract technology, as well as all the supporting computer science literature. Next, a plan was devised and carried out, culminating in a functioning Onboarding proof-of-concept established atop a blockchain-enhanced architecture. Finally, we evaluated the project in terms of performance and concept, which were both ultimately deemed viable. Having accomplished all of its goals, this thesis was considered a success by both Novabase and the writer.

6.2 Research Findings

Banks - and other businesses in the financial services industry - place incredible emphasis on assuring the security of their processes and data, as even ordinary transactions may involve large sums of money and resources. As customer interaction in banking becomes more directed towards digital strategies, special attention should be paid to the choice of back end systems that serve as the backbone to the digital channel.

This investigation concluded that blockchain is a valuable asset that banks have more than enough reasons to embrace. Most importantly, data immutability is a desirable property to include in the onboarding process, as the authenticity and auditability of the account contract are paramount to both bank and customer.

The smart contract, however, has yet to mature and develop an identity of its own separate from the basic digital contract. As it stands, use cases like the one explored seem to reap no more benefits from smart contracts than from the systems already in place.

Nevertheless, smart contract business logic is still an integral component of permissioned blockchain and will more often than not be used in conjunction with it. Furthermore, banks may find value in having a much lighter blockchain system handle onboarding, instead of the banking core.

6.3 Final Thoughts

Blockchain and smart contracts are fresh technologies with a long way to go before widespread adoption. However, the advantages in security they bring are apparent and relevant to all businesses - not only banks. It is often argued that the future is digital, cliché as though it may sound, and it is plausible that blockchain will play a role in defining the security of that future.

6.4 Further Developments

The final section of this document covers a short deliberation about further evolution and applicability of the body of work and research that was accomplished.

6.4.1 Customizable Smart Contract

Evidently, the variety of possible clauses that can compose an account opening contract is not limited to what this project took into consideration.

Let us consider an online account opening request scenario where the creation of the contract precedes, and therefore is detached from, most verification procedures. For instance, the process may be blocked until the client submits key information that proves eligibility conditions such as the absence of a history of overdrafts, an empty criminal record or a trustworthy identity. Specific bank account types may require taxpayer identification, a proof of studying, or a residency card.

Realistic account opening processes are indeed intricate and must be tailored to the individual preferences of both the bank and the client. This requires an extensible smart contract paradigm where the chaincode supports a multitude of possible conditions, perhaps coupled with a drag-and-drop or checkbox-based front end interface for a friendly end-user experience.

6.4.2 Escrow Contracts

Escrow arrangements are established when two or more parties that wish to engage in a trade of money or goods bestow those assets upon a trusted third-party - the escrow agent - that distributes them accordingly.

Rather than transacting directly, each party consigns their items to the escrow agent, which holds them until the trade is reciprocated fully. Should any other party not deliver on their part of the deal, the escrow agent safely returns any held assets. The escrow agent acts as a neutral mediator that enforces the deal by only enacting the transaction if all parties comply with it as established.

Escrows are a useful way of conducting a transaction when the two dealing entities do not trust each other. However, they do have to place their trust upon the neutrality and legitimacy of a middleman, which only shifts the problem around. Just as any party can fail to adhere to the negotiation, so too can the escrow agent.

Whereas a traditional escrow agent is an entity capable of free will, a smart contract running on a blockchain is a deterministic automaton incapable of negligence or malice. The final envisioned application is thus a trustless form of escrow - one which is made possible by the fact that the smart contract eliminates the need for a trusted third-party.

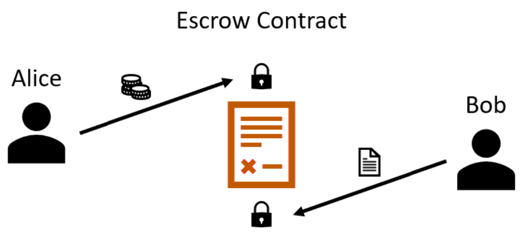


Figure 6.1: The escrow contract holds all assets delivered to it until all its clauses are fulfilled.

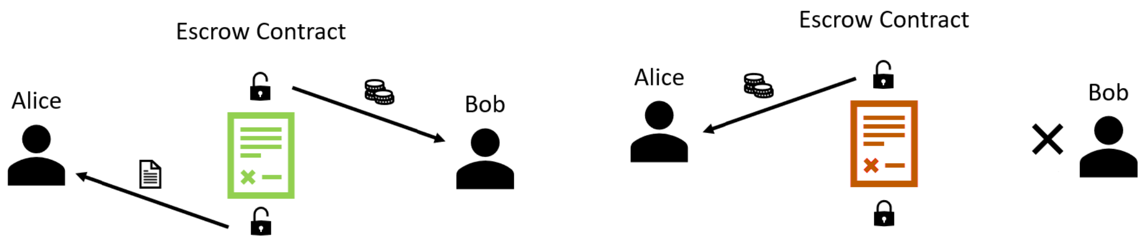


Figure 6.2: If all parties abide by the contract, the retained items are disbursed; otherwise, each item is restored to the respective authentic party that sent it.

Bibliography

[1] url: https://www.hyperledger.org/wp-content/uploads/2017/08/Hyperledger_Arch_WG_Paper_1_Consensus.pdf.

[2] url: https://www.hyperledger.org/wp-content/uploads/2018/04/Hyperledger_Arch_WG_Paper_2_SmartContracts.pdf.

[3] Imran Bashir. *Mastering Blockchain: Distributed ledger technology, decentralization, and smart contracts explained*. Packt Publishing Ltd, 2018.

[4] Miguel Castro and Barbara Liskov. "Practical Byzantine fault tolerance and proactive recovery." In: *ACM Transactions on Computer Systems (TOCS)* 20.4 (2002), pp. 398–461.

[5] George F Coulouris, Jean Dollimore, and Tim Kindberg. *Distributed systems: concepts and design*. Pearson Education, 2005. url: https://ce.guilan.ac.ir/images/other/soft/distribdystems.pdf.

[6] Jonathan Katz and Yehuda Lindell. *Introduction to modern cryptography*. Chapman and Hall/CRC, 2014.

[7] Sunny King and Scott Nadal. "Ppcoin: Peer-to-peer crypto-currency with proof-of-stake." In: *self-published paper, August* 19 (2012). url: https://www.peercoin.net/whitepapers/peercoin-paper.pdf.

[8] Leslie Lamport, Robert Shostak, and Marshall Pease. "The Byzantine generals problem." In: *Concurrency: the Works of Leslie Lamport*. 2019, pp. 203–226.

[9] Satoshi Nakamoto. *Bitcoin: A peer-to-peer electronic cash system*. Tech. rep. Manubot, 2019.

[10] M Tamer Özsu and Patrick Valduriez. *Principles of distributed database systems*. Vol. 2. Springer, 1999.

[11] Rüdiger Schollmeier. "A definition of peer-to-peer networking for the classification of peer-to-peer architectures and applications." In: *Proceedings First International Conference on Peer-to-Peer Computing*. IEEE. 2001, pp. 101–102.

[12] Nick Szabo. "Formalizing and securing relationships on public networks." In: *First Monday* 2.9 (1997). url: https://firstmonday.org/ojs/index.php/fm/article/view/548/469.

[13] Parth Thakkar, Senthil Nathan, and Balaji Viswanathan. “Performance benchmarking and optimizing hyperledger fabric blockchain platform.” In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. IEEE. 2018, pp. 264–276.