Informatics

Towards Platform-Agnostic Smart Contracts

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Rene Raab, BSc
Matrikelnummer 01225391

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: A.o. Univ. Prof. Dr. Dipl.Ing. Eva Maria Kühn
Mitwirkung: Dr. Dipl.-Ing. Gerson Joskowicz

Wien, 1. September 2021

Rene Raab	Eva Maria Kühn

Informatics

Towards Platform-Agnostic Smart Contracts

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Rene Raab, BSc
Registration Number 01225391

to the Faculty of Informatics

at the TU Wien

Advisor: A.o. Univ. Prof. Dr. Dipl.Ing. Eva Maria Kühn
Assistance: Dr. Dipl.-Ing. Gerson Joskowicz

Vienna, 1st September, 2021

Rene Raab	Eva Maria Kühn

Erklärung zur Verfassung der Arbeit

Rene Raab, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 1. September 2021

Rene Raab

Danksagung

An dieser Stelle möchte ich mich bei all denjenigen bedanken, die mich während der Anfertigung dieser Masterarbeit unterstützt und motiviert haben.

Zuerst gebührt mein Dank Frau Prof. Eva Maria Kühn und Dr. Gerson Joskowicz, die meine Masterarbeit betreut und begutachtet haben. Viele ertragreiche Diskussionen und konstruktive Kritik haben zur Erstellung dieser Masterarbeit erheblich beigetragen. Dafür möchte ich mich herzlich bedanken.

Ein besonderer Dank kommt auch meiner Freundin zugute, welche während der Verfassung dieser Arbeit nie die Geduld mit mir verloren hat, obwohl es manchmal durchaus verständlich gewesen wäre.

Abschließend möchte ich mich bei meinen Eltern bedanken, die mir mein Studium durch ihre Unterstützung ermöglicht haben und stets ein offenes Ohr für mich hatten.

Acknowledgements

At this point, I want to thank everyone who supported and motivated me during the creation of this Master's thesis.

First, I want to thank Prof. Eva Maria Kühn and Dr. Gerson Joskowicz, who guided and assessed this Master's thesis. Numerous fruitful discussions and constructive feedback aided the creation of this work substantially. I want to thank you very much for that.

I also want to especially thank my girlfriend, who never lost her patience with me during the writing of this Master's thesis, although it would have been understandable in some cases.

Finally, I want to thank my parents, who made my study possible through their continued support. They always had an open ear for me.

Kurzfassung

Durch den Erfolg von Kryptowährungen über die letzten Jahre ist auch das Interesse an der darunterliegenden Technologie gestiegen. Wissenschaftler aus verschiedensten Bereichen untersuchen, ob Blockchains für ihre Zwecke verwendet werden können. Dieses große Forschungsinteresse sorgt für schnelle Entwicklung und damit auch für rasche Veränderung. Dies hindert vor allem im kommerziellen Bereich die Adaption von Blockchains, da hohe Risiken mit der Wahl einer nicht zukunftsträchtigen Technologie verbunden sind. Ein Weg um die Adaption von Blockchain-Technologie voranzutreiben und das damit verbundene Risiko zu mindern, ist die Entwicklungsarbeit für verteilte Applikationen plattformunabhängig zu gestalten. Aus diesem Grund hat sich diese Arbeit das Ziel gesetzt zu zeigen, dass es möglich ist, Smart Contracts unabhängig von der darunterliegenden Plattform zu definieren und anzusprechen. Um dieses Ziel zu erreichen wurde die Ledger Access (LA) API entwickelt. Die LA API enthält ein Programmiermodell für Smart Contracts welches es ermöglicht, das Datenmodell, die Geschäftslogik und die Konsensregeln zu definieren. Die Ausführung eines solchen Kontraktes funktioniert über eine Abstraktion der Client APIs der unterstützten Plattformen. Ein universeller Kontrakt, welcher im Zielnetzwerk installiert ist, nimmt die Kommandos über die Client API entgegen und führt die gewünschte Transaktion aus. Es wurden zwei Plattform ausgewählt, welche stellvertretend für die breite Bandbreite und verfügbaren Plattformen stehen. Hyperledger Fabric und Corda sind beide beliebte Plattformen, welche zwar ähnliche Ziele verfolgen, sich allerdings in vielen Bereichen stark unterscheiden. Die Kompatibilität von LA API mit diesen beiden Plattformen zeigt, dass die LA API mit verschiedensten Plattformen verwendet werden kann. Zusätzlich wurde eine Referenz-Implementierung erstellt, welche anstatt einer Blockchain eine SQL-Datenbank als Datenspeicher verwendet. Die Referenz-Implementierung wurde verwendet, um die Plattform-Integrationen für Corda und Fabric zu evaluieren. Dabei wird davon ausgegangen, dass die Referenz-Implementierung immer das richtige Ergebnis liefert. Die Corda und Fabric Implementierungen müssen unter allen Umständen das gleiche Ergebnis liefern, um zu zeigen, dass sie korrekt implementiert wurden und die vorgegebenen Anforderung erfüllen. Es wurde eine Reihenfolge von Interaktionen definiert welche alle zuvor erhobenen Anforderungen abdecken. Diese Interaktionen wurden dann auf allen drei Plattformen (Referenz, Corda, Fabric) ausgeführt. Ein Vergleich der Ergebnisse hat ergeben, dass alle Implementierungen das gleiche Ergebnis liefern. Durch diesen Proof of Concept wurde bestätigt, dass plattformunabhängige Smart Contracts möglich sind.

Abstract

The success of cryptocurrency over the last few years has generated a lot of interest in the underlying technology. Scientists from various research areas are trying to adopt blockchain technology. This research interest drives rapid development and causes frequent technological changes. Commercial adoption of blockchain technology is currently limited because adopting a technology that may not be successful brings a lot of risk for enterprises. One way to reduce this risk and advance the adaption of blockchain technology is to make the development of distributed blockchain applications platform-independent. For this reason, the goal of this work was to show that it is possible to develop and execute smart contracts independent from the underlying target platform. To reach this goal, the Ledger Access (LA) API was developed. The LA API provides a programming model that allows the data model, the business logic, and the consensus rules of smart contracts to be defined independent from the underlying platform. The execution of such contracts works by utilizing an abstraction over the client APIs of the supported platforms. A universal contract installed on the target platform receives commands over the client API and executes the respective transaction. Two different platforms representative of the wide variety of blockchain platforms on the market were selected. Hyperledger Fabric and Corda are both widely used platforms that follow similar goals, but are different in many aspects. The compatibility of the LA API with these two platforms shows that the LA API can be used with a wide range of platforms. In addition, a reference implementation that uses an SQL database instead of a blockchain in the background was created. The reference implementation was used to evaluate the presented platform integrations for Fabric and Corda. For that, it is assumed that the reference implementation always returns the correct result. All other adapters have to return the same result under all circumstances. To test the functionality of the integrations, a use case was designed and implemented. A scenario containing all the important requirements was developed from the use case. This scenario was executed on all three platforms (Reference, Fabric, and Corda) and the results were compared. The comparison showed that all executions on all platforms delivered the same results. This proof-of-concept confirms that it is possible to create and execute platform-agnostic smart contracts.

Contents

CHAPTER 1

Introduction

With the rise of cryptocurrencies like Bitcoin and Ethereum over the last decade, the underlying technologies became popular research topics in various scientific disciplines. Especially computer scientists and economic researchers try to apply and optimize blockchain technologies like smart contracts and electronic cash across various domains. But scientists are not the only ones interested in finding new ways to use this technology in all kinds of different areas and applications. Regardless of whether talking about small start-ups or big corporations, many innovative enterprises are trying to adopt this technology.

The big promise of blockchain technology is decentralized trust. Most transactions made today, no matter if it is wiring money or buying a house, require the involvement of a centralized institution like a bank to establish trust between the transacting parties. This means that these centralized institutions must be trusted by everyone involved, but the transacting parties do not need to trust each other. Decentralized trust means that such an institution is not necessary for parties that do not trust each other to transact with each other.

Since there is such a broad area of promising applications for decentralized trust, many variations of blockchain technology evolved. Each one has its advantages and disadvantages. Although blockchain implementations can be very different, they all share the same basic functionality. A network of nodes manages some kind of distributed state, together with the whole history of all changes ever made to the state. The challenge is to reach a consensus about changes to the state without having to trust a central authority.

Public permissionless blockchains are the most widely known type of blockchains. They are utilized by cryptocurrencies like Bitcoin and Etherum. The main characteristic of public blockchains is that everyone with a computation device and internet access can join the network and start participating in all activities (mining, transaction validation...) of the blockchain network. This includes producing coins, validating transactions, and validating blocks. But decentralized digital money is not the only potentially revolutionary use case for blockchain technology. One other example where this technology could change the way things are done fundamentally is day-to-day business-to-business transactions. Contrary to digital money, where it is desired that everyone can join the network and have equal rights, in a business setting fine-grained permissions are necessary to model the hierarchy and transaction approval systems of a company. For this reason, permissioned blockchains, also called distributed ledger technologies (DLTs), were created.

DLTs require each node to be authenticated and known to join the network. Most implementations use digital certificates issued by a trusted certificate authority to identify a node and determine its company affiliation. The data saved in the distributed ledger, also referred to as its state, is accessed and changed by running smart contract functions. A smart contract is a piece of code, located on the blockchain, which defines the allowed state transitions. Allowed means in this context that the required entities have agreed that the code in the contract defines exactly how interactions with the distributed ledger should be executed.

1.1 Problem Statement

Distributed ledgers are still a new technology and are under intensive development. This means that their APIs are unstable and subject to change. Moreover, it is still not clear which platforms, concepts, and approaches will become a success and which will fail to be useful in the real world. Another factor is that different platforms have different trade-offs. This means that changing requirements might make it necessary to change platforms in the future, even if the chosen platform is a success. The point is that there are numerous reasons why a platform change might become necessary in the future. Such a change would be very expensive, because all the developed smart contracts and client applications would have to be redeveloped from scratch. These uncertainties increase the risk of committing to a platform and therefore limit the adoption rate of DLTs. For this reason, this work aims to develop a framework that allows for the definition and execution of smart contracts independent from the underlying platform.

1.2 Aim of the Work

The goal of this work is to design an API that enables developers to create smart contracts and interact with them in a platform-agnostic way. Such an API would solve the problems explained in the problem statement, as switching platforms would be much less expensive. Two main parts are necessary to provide the described functionality. First, a generalized interface that allows transactions to be submitted in a platform-independent way will be designed. For this to work, each supported target blockchain installation needs to have a general smart contract that is used by the API to send and receive information to/from the ledger to be installed. Initially, this API will support two platforms, namely Hyperledger Fabric[1] and Corda[2]. There will also be a mock-up that helps the development of smart contracts and serves as the reference implementation. In other words, the implementations for blockchain platforms must, under all circumstances, deliver the same results as the mock-up implementation. This reference implementation will be a single node with a simple relational database to store the state. Second, a programming model that allows for the implementation of platform-agnostic smart contracts will be designed. This model, and therefore the contracts implemented with it, will make use of the proposed API to trigger transactions and retrieve current state information. To show the viability of the API and the programming model, a sample contract will be implemented. This contract allows for the management of the lifecycle of various assets. The supported operations include creation, transfer, and atomic exchange of assets. These operations correlate 1:1 to the trading-based community blockchain interaction patterns identified by M. Sengstschmid in [Sen19]. A test scenario with multiple transactions will be designed to show that all platforms deliver the same results for the same sequence of transactions. It is also worth noting that the mock-up or reference implementation makes it possible to write and test contract logic without the need for a complex test environment with multiple nodes. To sum up, the result of this work will be a prototype implementation of the proposed API design. The prototype will allow smart contracts to be designed and executed on either a Corda network, a Hyperledger Fabric network, or the mock-up implementation.

1.3 Methodological Approach

The first step is to perform a literature review that covers two topics. The first part is an analysis of popular permissioned blockchain implementations like Fabric and Corda. The second topic covers the state of the art in platform-independent smart contract deployment and execution. The goal of researching these two topics is to identify concepts, technologies, and patterns that can be used to design a versatile model for developing smart contracts once and run them on multiple platforms. Furthermore, the literature review is used to ensure that the proposed solution fills an existing research gap.

The next step is to define the requirements of a platform-agnostic smart contract framework in the form of user stories. This involves describing the involved stakeholder and formulating the requirements from their point of view.

After that, a framework that allows platform-agnostic smart contracts to be created and executed on different platforms will be designed. The programming model will be designed and presented by utilizing commonly used software architecture tools and methods. Software patterns will be used to create a model which is easily understandable

[1]https://www.hyperledger.org/use/fabric
[2]https://www.corda.net/

because it uses familiar concepts. In addition, the model will be presented to the audience with the help of UML diagrams.

With the knowledge gained from the literature review, the defined requirements, and the definition of the programming model, a prototype can be implemented. The prototype should show that the model supports the defined use case and that the developed contract can be deployed and executed on multiple blockchain platforms. Corda and Hyperledger Fabric will be the target platforms for this proof-of-concept implementation. A reference implementation will additionally be used to define the expected behavior of the real platforms. This will become useful for the evaluation, as well as for developing the contract logic.

Finally, the proposed solution will be evaluated to show that its design is versatile enough to fulfill all requirements and that the implemented prototype works as expected. This will be done in two steps. First, it will be shown that the design supports all functional and non-functional requirements defined by the user stories. The second step is a functional evaluation of the prototype. A test scenario with multiple transactions will be defined and implemented. The aforementioned reference implementation delivers the reference results for the test cases. After that, the test cases will be executed on the selected platforms and the results compared. If the results match, it is shown that the model supports the defined use case and that its possible to execute contracts developed according to the model on different blockchain platforms. Execution on different platforms requires having a test network with multiple nodes available. Containerization in the form of Docker makes it possible to run such a network quickly, cheaply, and on one local machine.

1.4 Structure of the Work

This thesis is split into eleven chapters. The first chapter (1) contains an introduction to the topic. This includes the problem statement, the aim of the work, and a description of the methodological approach. Chapter 2 gives an overview of the theoretical background of the work. This is mainly an introduction to blockchain as a whole, as well as the specific permissioned blockchain platforms that will be used later in this work. After that, Chapter 3 gives an overview of the related work of this thesis. This will cover the most important and influential tasks in the area of platform-agnostic smart contracts. A comparison of the solutions found in the literature will show that no existing approach fulfills all our requirements and that the proposed work therefore presents a novel approach to platform-independent smart contracts. Chapter 4 formulates the requirements for a platform-agnostic smart-contract framework in the form of user stories. In Chapter 5 the design of the *LA API* is introduced. The *LA API* is the core of the proposed solution and will enable the definition and execution of smart contracts in a platform-independent way. In Chapter 6 a fictitious smart contract use case which will aid in the evaluation of the *LA API* is introduced. The following chapters are mainly focused on providing a fully functional, end-to-end prototype implementation of a smart contract that can be executed on multiple platforms. Chapter 7 describes the technologies and tools that will be used to create the prototype. Chapter 8 shows how distributed ledger platforms can be integrated into the *LA API*. This will include sample implementations for *Corda* and *Fabric*. Next, Chapter 9 uses the *LA API* to implement the use case defined in Chapter 6. The implementation part of the thesis is concluded in Chapter 10 with a sample implementation for a client, a program that uses the *LA API* to execute platform-agnostic smart contracts. An evaluation of the *LA API* is made in Chapter 11 by showing that all user stories are implemented and that a functional test covering all use cases and platforms succeeds. After that, Chapter 12 details some open questions and a path on how they can be answered in future work. The thesis is concluded in Chapter 13 by giving a summary of the work and its findings.

CHAPTER 2

Blockchain Basics

Blockchain technology was first proposed in 2009 by an anonymous author who used the pseudo-name Satoshi Nakamoto[Nak19]. Nakamoto introduced a digital cash system named Bitcoin that allows parties to transfer money anonymously over the internet without the oversight of a third party. At its core Bitcoin uses a blockchain, sometimes also called distributed ledger, that can be described as an append-only data structure managed by an arbitrary amount of contributing nodes in a peer-to-peer (P2P) network. In traditional paper money systems, digital transactions require a central authority (e.g., a bank) trusted by all involved parties to validate that the money in question exists and to avoid double-spending. Bitcoin removes this constraint by introducing a distributed consensus mechanism which controls how new entries (or blocks) can be appended to the distributed ledger. This is the core advantage of all blockchain technologies: it is no longer necessary to establish trust between interacting parties. Furthermore, most blockchain implementations also can tolerate participants acting in nefarious ways. Not relying on a central authority and being able to interact with untrusted parties is not only helpful in designing money systems but can also be used in various other domains. This lead to a multitude of different blockchain implementations with different fields of applications and design goals. In general, blockchains can be categorized as public or private (also permissioned) blockchains. Cryptocurrencies like Bitcoin are public, which means that all the data is publicly accessible and everyone can join the network without providing any proof of identity. Other domains, like supply chain management, require participants to be authenticated and the stored data and transactions should not be visible to everyone. For these applications, private/permissioned blockchains are used. An example of a popular permissioned blockchain implementation is Hyperledger Fabric[1].

[1]https://www.hyperledger.org/use/fabric

2.1 Hash Functions

Hash functions are mathematical functions that map an arbitrary amount of data to a fixed-size value. The resulting value is called the hash value. Hash functions are widely used to store passwords and to verify that data is not corrupted or changed. A hash function needs to fulfill the following properties to be considered secure:

- **Non-reversibility**: It should be practically infeasible to calculate the original data from the hash value.
- **Non-predictable**: It should not be possible to predict the hash value when given specific data.
- **Determinism**: Calculating the hash value of given data should always return the same result.
- **Collision resistance**: It should be practically infeasible to find different data sets that result in the same hash value.
- **Diffusion**: A small change in the data should result in a big and unpredictable change of the hash value.

Bitcoin and other blockchain implementations use a hash function to ensure the immutability of the transaction history. How the properties of secure hash functions are used to ensure the safe and verifiable operation of blockchains is discussed in later sections of this chapter.

2.2 Asymmetric Cryptography

Asymmetric Cryptography describes encryption techniques that make use of a key pair. A key pair consists of a private and a public key. This is why these encryption systems are often also referred to as public-key encryption. The basic principle is that the private key is only known to the user and the public key can be distributed publicly. The two keys are mathematically related in a way that makes it possible to encrypt with one key and decrypt with the other. In theory, it is possible to calculate the private key from the public key. In practice, this calculation is infeasible and would take multiple lifetimes even with the most powerful supercomputers available today. Encrypting a message with the public key ensures that only the person in possession of the private key can read it. When a message is encrypted with a private key it can be decrypted with the public key. Since only the person in possession of the private key can do that, everyone can verify from whom the message is. This process is also known as signing or digital signature.

2.3 Network

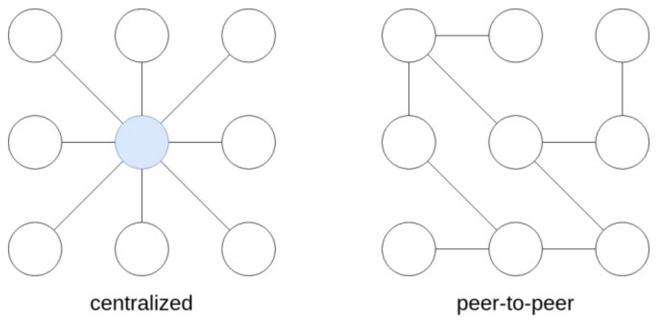

Figure 2.1: Difference between a centralized and a P2P network topology. In the centralized approach (left), all clients communicate with a single server (blue node). In a P2P network (right), there is no central server and nodes communicate directly with each other.

The network over which an application communicates can be viewed on two different layers of abstraction. One describes the physical package flow from one node to the other. Such a network description involves hardware like switches and routers. For this work, we only look at the overlay network. An overlay network abstracts the hardware details of the physical network away and describes it from the application point of view. There are two major network topologies used in modern networking applications. The first one is a centralized approach where multiple different clients talk to a single server. Having a centralized network has a wide variety of advantages and disadvantages. One major drawback is that it introduces a single point of failure, which makes it vulnerable to denial of service (DoS) attacks and network outages. Horizontal scaling, meaning adding additional computing power by adding new servers, is another weak point of a centralized approach. Its wide usage stems from the fact that it is much simpler to implement than the second type of network, which is called a peer-to-peer network. Peer-to-peer networks do not rely on a central server, but the work being done is distributed to multiple nodes. To make such a network work, mechanisms for discovering new nodes and sharing the work have to be in place. This is why P2P networks are more complex compared to centralized approaches. [Sch01] define a P2P network as follows:

> ”A distributed network architecture may be called a Peer-to-Peer (P-to-P, P2P, ...) network, if the participants share a part of their hardware resources (processing power, storage capacity, network link capacity, printers, ...). These shared resources are necessary to provide the Service and content offered by the network (e.g. file-sharing or shared workspaces for collaboration): They are accessible by other peers directly, without passing intermediary entities. The participants of such a network are thus resource (Service and content) providers as well as resource (Service and content) requestors (Servent-concept).”

Since blockchains are specifically designed to have no centralized authority, all implementations use a P2P overlay network for communication between peers.

2.4 Generic Blockchain Operation Flow

This section gives a general overview of the inner workings of a blockchain by describing a simplified version of the Bitcoin protocol. The following sections then describe each component in more detail. Since blockchains are append-only data structures there is no way to change existing data; only the state of the whole chain can be changed. This is done by submitting a transaction to the network. A transaction describes the transition from one state of the blockchain to the next. Once the transaction is received by a node in the P2P network, it is broadcasted to all the other nodes. Mining nodes collect valid transactions and bundle them into a block. Each block has the hash of the previous node saved as a reference. This ensures that the chain cannot be changed. The final step of building a block involves solving a cryptographic puzzle, which is also known as Proof-of-Work (PoW). Once this puzzle is solved, the block is propagated to the other nodes in the chain. The other participants validate the new block. If the PoW of different blocks is solved at similar times, the longest chain rule applies. At some point two competing blocks have to have the same parent; the longest chain rule states that the block with the most intermediary blocks to the common parent block is valid/part of the main chain and the other block is invalid/not part of the main chain. As a rule of thumb, it can be assumed that once six blocks are mined on top of a block, the block is confirmed and therefore part of the blockchain. The 6 blocks rule is based on statistical probability and therefore not absolute. Blocks that are not part of the main chain, but a branch of the chain, are called orphan or stale blocks.

2.5 Bitcoin

This section describes the Bitcoin protocol and its components in more detail. As the first and most popular cryptocurrency, Bitcoin can be considered as the foundation of all blockchains. Since it is not a perfect protocol and has many weak points, improvements to Bitcoin have been proposed in the literature. The most important of these new approaches will be discussed in a later section of this chapter.

2.5.1 Nodes

Peers in the P2P overlay network in blockchains are also called nodes. Using the dynamic properties of the P2P network, nodes can join and leave the network at any time. The Bitcoin network is fully democratic, which means that every node has the same rights as every other node. There are a few different types of nodes in a Bitcoin network. They are distinguished by the tasks they are executing in the network. For the basic understanding, this work is trying to convey it is sufficient to look at two basic types, which are full nodes and mining nodes.

Full nodes are vital for the security of the system because they validate transactions and blocks. They also relay blocks and transactions and are therefore responsible for propagating changes to other nodes. A full node also holds a full copy of the blockchain locally, which is downloaded initially from other full nodes in the network and then maintained according to the consensus mechanism (see Section 2.5.5).

Mining nodes are nodes that contribute their computing power to participate in the mining process. In Bitcoin, mining is another word for solving the PoW puzzle. Miners can work alone (solo miners) or bundle their resources with other miners (pool miners). Since contributing computing power is expensive, Bitcoin and other cryptocurrencies have an integrated incentive system. A miner gets rewarded with a certain amount of coins if he/she mines a block and the block becomes part of the blockchain. The amount of the reward varies over time and is halved roughly every four years. In the year 2020, mining a block in the Bitcoin network was rewarded with 6.25 coins.

2.5.2 Transactions

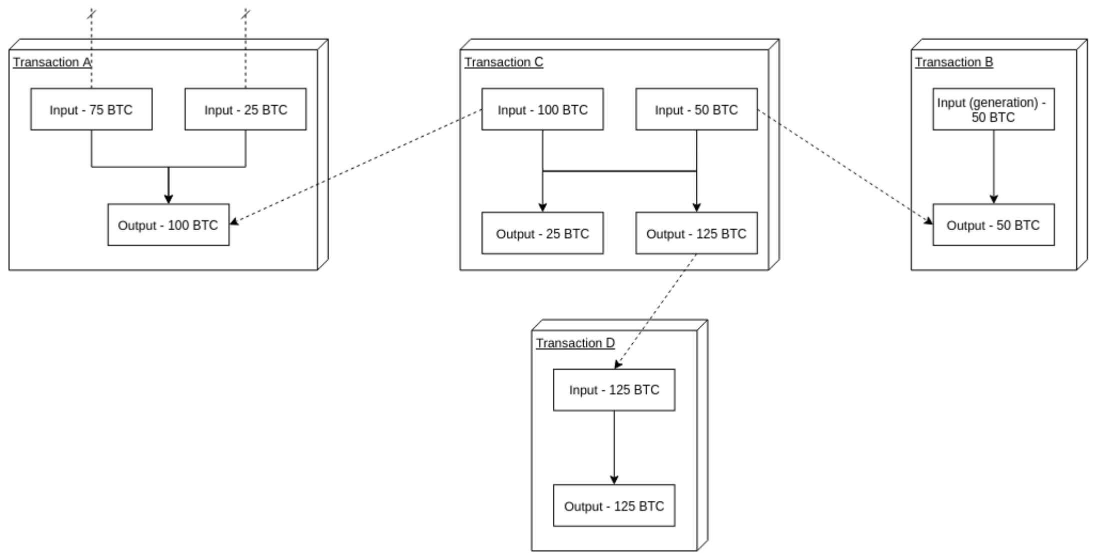

Figure 2.2: Example of a UTXO transaction flow

Blockchains represent an append-only data structure, which means that there is no way to change existing data. At a point in time, all the data contained in the blockchain represents a specific state. Moving the chain from the current to the next state is done by adding blocks at the end of the chain. Each block in the chain contains multiple transactions which describe a state transition. In the case of Bitcoin, transactions describe the transfer of coins from one wallet address to another. Each transaction can have one or more inputs and one or more outputs. A transaction input can either be an unspent transaction output (UTXO) from a previous transaction or be generated through the mining process. The concept of multiple inputs can be seen as using multiple coins in cash transactions. Having multiple outputs allows a user to send coins to multiple recipients inside one transaction. The sum of all outputs has to be smaller or equal to the sum of all inputs for a transaction to be valid. Any input not spent on an output is considered a transaction fee and can be claimed by the miner. If a user wants to transfer only part of the coins from one input to another user, an additional output, where the change is returned to the instigating user, has to be added.

Figure 2.2 shows a flow of coins through multiple transactions. Transaction A has two inputs from previously unspent transaction outputs and combines them into one output containing 100 coins (BTC). Transaction C uses these 100 BTC and the 50 BTC generated through mining from Transaction B to transfer 125 BTC to another user. The second output of Transaction C represents the change of the transaction, i.e., the amount of BTC the peer receives back. Transaction D uses the UTXO from Transaction C and transfers all of it to another user. All the shown transactions are valid because the sum of the inputs is always equal to the sum of the outputs.

2.5.3 Wallets

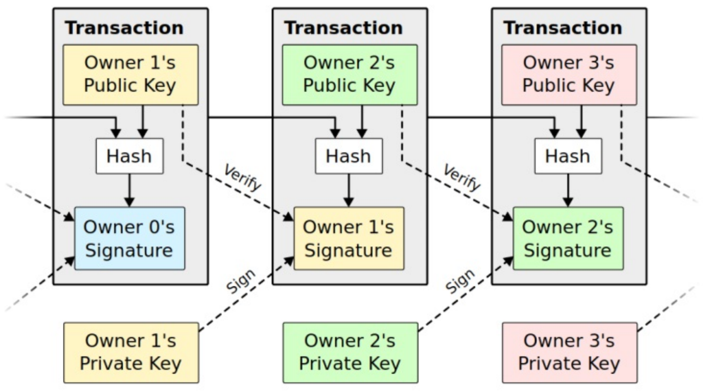

Figure 2.3: Transaction signing and verification process [Nak19]

Wallets are often misconstrued as the place where coins of a cryptocurrency are stored. Due to the distributed nature of blockchain, coins are not stored in one place. They are part of the distributed data structure managed by the participating nodes. A wallet stores the credentials which are used to verify that the UTXO from a previous transaction belongs to a specific user. At its core, a wallet is nothing more than a cryptographic public/private key pair. This key pair is used to sign and encrypt transactions. In addition to the key pair, a wallet also has an address. This address corresponds to the hashed value of the public key and is used as the identifier of a wallet.

Figure 2.3 shows how asymmetric cryptography is used to sign and verify transactions in Bitcoin. The owner of a coin calculates a hash from the next owner's public key and the previous transaction. Then the owner signs this hash value with his/her private key. Everyone can now verify the current owner with the public key of the previous owner. Therefore, the current owner can take a public key of the next user and start the process from the beginning to transfer the coin to the next user. Since the hash of the previous transaction is also signed, a chain of transactions builds. This makes it possible to track all owners of a coin back to its creation.

2.5.4 Data Structure

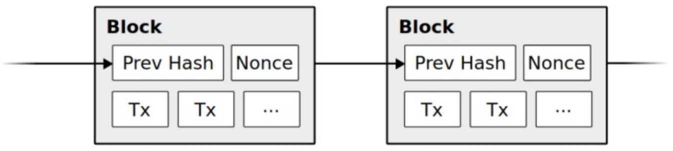

Figure 2.4: Example of a blockchain segment with two blocks [Nak19]

As the name suggests, a blockchain is a chain of blocks. At its core, a blockchain is nothing more than a linked list where each element is a block that contains multiple transactions. The links between blocks are established by including the hash value of the previous block in the header of the next block. Figure 2.4 shows a section of a blockchain with two blocks. Besides the hash of the previous block, a block also contains several transactions (see Section 2.5.2) and a nonce value. The nonce is used in the calculation of the PoW puzzle (see Section 2.5.5).

The properties of cryptographic hash functions ensure that once a block is linked to another block it is not possible to change it (see Section 2.1). The reasoning behind this immutability guarantee is best described by the following example. Let's assume we have two blocks, A and B, with block A being the predecessor of block B. This means that block B contains the hash value of block A. A change in block A would lead to a change in its hash value. This change can easily be detected by comparing the hash value stored in block B with its current value. Combining this hashing procedure with the longest chain rule ensures that the data contained in the blockchain is immutable after a certain number of blocks are added on top of it. The longest chain rule states that the chain with the highest amount of blocks is the current valid chain. It should be noted that this immutability guarantee stems from the fact that it is statistically and computationally not feasible to recalculate the blocks and build a longer chain. Therefore, the guarantee is not absolute.

The first block of a blockchain is called the genesis block and is normally hard-coded. Since the genesis block does not point to any prior block, the previous hash field contains a null value.

2.5.5 Consensus

The consensus algorithm of a blockchain dictates how peers come to an agreement over which block is added to the blockchain next. It allows the peers to establish trust without the need for a central authority. In Bitcoin and many other public blockchains, a proof-of-work is used as a kind of majority voting system. PoW was first introduced in Adam Backs's Hashcash [B+02] to combat denial-of-service attacks. Blockchains use it to establish that each mined block used some amount of computing power in its creation. The longest chain of mined blocks is the one with the most computing power in it, which represents the majority decision of the network.

The PoW algorithm is, at its core, very simple and can be described as follows. A miner collects transactions into a block and solves a puzzle. This puzzle is to find an SHA-256 hash value with a defined number of zeros at the beginning. The amount of leading zeros determines the difficulty of the puzzle. In Bitcoin, the difficulty of the puzzle is adjusted depending on the total computing power of the whole network. The target is to produce one valid proof-of-work networkwide roughly every ten minutes. Since applying the hash function to the same data would always lead to the same hash value, the input has to be changed. This is what the nonce field in a block is for (see Figure 2.4). Mining a block is the process of changing the nonce value and recalculating the hash value until one is found which fulfills the set difficulty.

Although PoW is widely used it is not an ideal solution. One problem is that the amount of transactions per minute is not scalable enough to support wide adaption. Another problem is energy consumption. A vast amount of energy is used to calculate the hashes. This energy is largely wasted, because there is no use for the results that do not fulfill the difficulty requirements, i.e., do not produce a valid proof-of-work. These problems lead to a desire for alternative consensus mechanisms. A few of these approaches will be discussed in the upcoming section of this chapter.

2.6 Ethereum

Ethereum is another well-known cryptocurrency based on blockchain technology. First introduced by Vitalik Buterin in 2014[B+14], Ethereum presented several innovative concepts to the distributed ledger field, its most important being smart contracts. This chapter will discuss these innovations and give an overview of the differences between Bitcoin and Ethereum.

2.6.1 Design Goals

The Ethereum white paper [B+14] defines design goals that are meant to guide the technical decisions necessary to further improve the platform. This section only provides a summary of the goals with the purpose of giving a sense of the overarching intentions of the Ethereum project. For a more detailed explanation, we refer to the white paper[2].

- **Simplicity**: Keeping the Ethereum protocol as simple as possible, even if that means sacrificing storage and time efficiencies.
- **Universality**: Instead of providing "features," Ethereum provides a Turing-complete programming language, which can be used by programmers to add additional functionality.
- **Modularity**: The components of the Ethereum protocol should be as modular and separable as possible.
- **Agility**: The Ethereum protocol should not be set in stone and should have the possibility of change incorporated.
- **Non-discrimination** and **non-censorship**: There should be no active attempt to prevent or restrict specific categories of usage.

2.6.2 Accounts

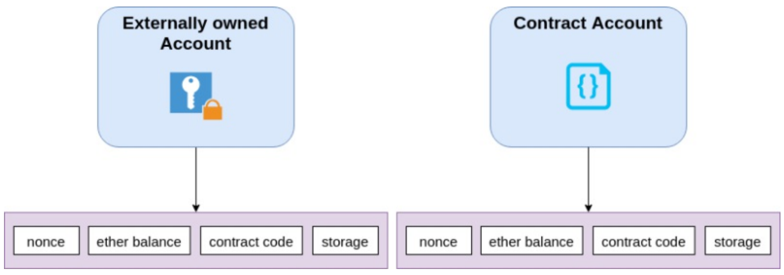

Figure 2.5: Structure and types of Ethereum accounts

In contrast to Bitcoin, the state of Ethereum is not represented by the UTXO, but stored in objects called "accounts." Ethereum distinguishes between two types of accounts (see Figure 2.5). Externally owned accounts are controlled by private keys and are similar to wallets in Bitcoin. Contract accounts are controlled by their associated contract code. Each account contains the following four fields: the nonce, the Ether balance, the contract code of the account, and the storage of the account. The nonce is a counter used to assure that a transaction cannot be processed more than once. The Ether balance field

[2]https://ethereum.org/en/whitepaper/

represents the amount of Ether held by the account. Ether is an internal currency used to pay transaction fees. For a contract account, the contract field contains a hash of the code associated with the account; for an externally owned account this field is empty. The storage field can be used by the contract code to read and write persistent information.

2.6.3 Messages and Transactions

Ethereum transactions are similar to transactions in Bitcoin, but they have three extra fields which are properties used for the execution of the contract. The two most important fields are called "STARTGAS" and "GASPRICE" and are the cornerstone of Ethereum's anti-denial-of-service strategy. Since Ethereum allows arbitrary remote code execution, an attacker could easily create a long-running contract code (e.g., infinite loop) and eat up valuable resources with no cost. To make such an attack expensive and therefore disincentivize it, each computational step executed has to be paid for. One computational step costs 1 "gas." The "STARTGAS" value contained in a transaction represents the maximum number of computational steps the execution is allowed to take. The "GASPRICE" represents the amount the sender is willing to pay for the transaction. The third field is a data field which can be written by the sender and read by the contract code the transaction targets.

The term "transaction" is only used for objects where the sender is an externally controlled account. For objects exchanged between contract accounts, the term "messages" is used. The format of messages and transactions is very similar, the only difference being that messages do not have the GASPRICE field. The reason is that the GASPRICE is always set by the account which initially sends a transaction to a contract. A message is produced when one contract calls another, which means that contracts can have relationships with each other. It is important to note that the gas allowance for the execution of a transaction must cover the execution of the contract and all the sub-contracts.

2.6.4 Ethereum Virtual Machine

The execution of the contract code happens in the Ethereum Virtual Machine (EVM). The EVM executes code written in a low-level, stack-based bytecode language. A program consists of a series of bytes, where each byte represents one operation. For the execution, the EVM runs an endless loop doing the following steps in each iteration. First, the value of the program counter (PC) is used to find the operation to execute in the code. This operation is then executed and the program counter is incremented. The execution of a command also decreases the GAS, to make sure each calculation is paid for. This loop is only stopped when all the GAS is consumed, an error is detected, or a finishing instruction like STOP is detected. EVM operations have access to three different types of memory: the stack, the memory, and the storage of the contract currently being executed. The contract storage is the only place where data can be stored for the long-term. The stack and memory are reset after the execution finishes.

2.6.5 Consensus

As of the writing of this thesis, Ehtereum uses a PoW consensus similar to Bitcoin, but there is a detailed roadmap with multiple defined phases to switch to proof-of-stake (PoS) consensus in the future[3]. The PoS system proposed by Ethereum is called Casper and a detailed explanation of its workings can be found in [BG19].

In a PoW, consensus nodes contribute computing power to solve a cryptographic puzzle. Solving the puzzle gives them the right to create a new block. In a PoS consensus, the right to create a new block is dependent on the size of the stake a node has in the process. The stake is defined as the number of coins locked away in a special contract. Each validator node that has a stake is added to a pool, which the blockchain uses to decide which block is added next. There are two major types of PoS that take a different approach to decide which block is confirmed next. Chain-based PoS algorithms select a node for each time slot pseudo-randomly. The size of the stake relative to the sum of all stakes invested determines the probability of being selected. Once a node is selected, it gets the right to create a single block. In contrast to that, a Byzantine-fault tolerant (BFT)-based PoS consensus uses a multi-round voting process to determine the next block. Similar to the chain-based approach, a node that is selected randomly gets the right to propose a block. All participants then send votes for the block. The key difference here is that once a block gets enough votes it is permanently confirmed. In other words, the confirmation of the block is instant and does not depend on the number of blocks on top of it.

PoS promises to solve many problems PoW-based blockchain implementations have been widely criticized for. Since PoS does not depend on solving a computationally expensive puzzle, the energy consumption of PoS-based blockchains is reduced by an order of magnitude compared to PoW blockchains. Because PoW requires so much electricity and expensive hardware to meaningfully participate, centralized structures, which combine their computing power in mining pools, are formed over time. This is a problem since it threatens the core promise of blockchain technology, which is to prefer distribution over centralization. PoS is expected to solve this problem since there are no disproportionate gains. If someone has \$10 million coins, the expected profit gained from staking is exactly ten times higher than of someone with \$1 million coins. The same is not true for miners in a PoW chain. PoS is also expected to provide additional security for known attacks like the selfish mining[BZW$^+$19] and the 51%[SMG19] attack.

2.7 Permissioned Blockchain Primer

In contrast to Bitcoin and Ethereum, participants of a permissioned (private) blockchain have to be authenticated and given privileges to execute actions on the blockchain. This

[3]https://docs.ethhub.io/ethereum-roadmap/ethereum-2.0/proof-of-stake/

concept opens DLTs up to be used in several different day-to-day business processes. Currently, there is no unified system that allows identity and transaction data to be managed over company boundaries that doesn't involve trusting each other or some third party. The consequence is that each company has its own system, and interacting with business partners (e.g., signing and executing contracts) is hard to automate and involves manual tasks like communication over e-mail. Permissioned DLTs try to solve this by providing a system that allows contracts and permissions to be programmatically defined in a transparent manner. Combined with the append-only nature of blockchains, interacting cooperations build a single transaction history (instead of everyone building his/her own) based on agreed-upon rules. This simplification of business transactions has the potential to hugely optimize and automate today's highly complicated and globalized economy.

2.8 Hyperledger Fabric

Hyperledger Fabric[4] is one of the most popular permissioned distributed ledgers technology implementation. It is developed under the Hyperledger[5] umbrella, which is an open-source community dedicated to delivering enterprise-grade frameworks, tools, and libraries for permissioned blockchains.

2.8.1 A Fabric Network and Its Components

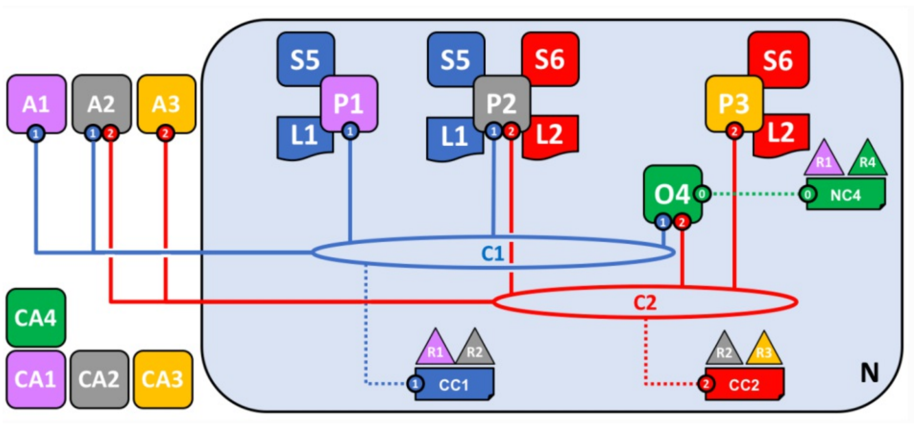

Figure 2.6: Example of a Hyperledger Fabric network taken from the official documentation[6]

[4]https://www.hyperledger.org/use/fabric
[5]https://www.hyperledger.org/
[6]https://hyperledger-fabric.readthedocs.io/en/release-2.2/network/network.html

A Hyperledger Fabric network is built by a group of organizations, called a consortium. The reason for creating such a network is that the organizations have some need to interact/transact with each other. A Fabric network lets them do that by sharing resources and operating inside the same system. The blockchain nature of Fabric allows a consortium to do that without the need to trust the partners inside the consortium or another central authority. All actions inside the network are authenticated and need to follow policies defined in the network. Policies are a core mechanism of Fabric, since they define who is allowed to request that an action should be taken and who has to agree on that action to execute it. On a logical level, it follows that participants of a Hyperledger Fabric network are organizations. These can be large, multinational conglomerates or a small bike repair shop around the corner. Figure 2.6 shows an abstract example of a Fabric network with all components needed for it to be fully functional.

The diagram contains the following entities. There are four organizations: R1, R2, R3, and R4. Colors in the diagram are important. Each component marked with the same color belongs to the same organization. For example, R3 is yellow, therefore A3, CA3, and P3 are operated by R3. A1, A2, and A3 are client applications. CA1, CA2, and CA3 are certificate authorities. P1, P2, and P3 are peers. Peers have contracts (S5, S6) installed, which maintain their respective ledgers (L1, L2). This relationship is also color-coded. For example, contract S5 maintains the ledger L1. Peers and client applications communicate over channels, which are governed by channel configurations denoted as CC1 and CC2. The triangles over configuration elements denote by which organizations the configuration is governed. The members of a channel are denoted by the lines going into them. A1, A2, P1, and P2 are all members of Channel C1. P3, on the other hand, is only a member of C2 and not C1.

At the beginning of every Fabric network the ordering service is created. In this example, the ordering service is represented by the single node (O4), operated by the organization R4. As the name suggests, the responsibility of the ordering service is to order (and verify) incoming transactions and put them into a block. Although it is only a single node here, Fabric supports the scaling of the ordering service over multiple nodes run by different member organizations. The ordering service is responsible for maintaining and enforcing the policies defined in the network configuration (NC4) and can be seen as the network administration point. In our example, the network configuration is governed by the organizations R1 and R4. Policies not only allow fine-grained control over who can execute certain actions in the network, but also define how policies themselves can be changed. For this example, we can assume that R1 and R4 have to agree on all changes to the network configuration, but this is not necessarily the case in real networks. It depends on the specific policies. Since policies are very flexible, one can imagine that R4 can execute actions without the approval of R1, and R1 always needs the approval of R4.

Once two or more cooperations decide they want to transact with each other, they form a consortium. Consortium definitions are saved in the network configuration. As already discussed, in the example, changes in the network configuration need the approval of R1 and R4. The consequence is that they are the only organizations in the network that can create consortiums, at least in this example under the assumptions we made earlier. Of course, this can be changed by adapting the policy.

Most likely the members of the consortium want to keep their interactions private. Fabric supports data privacy with the concept of channels. Each channel maintains a ledger and a set of smart contracts, which are used to change the state of the ledger. Only members of a channel have access to the ledger and the contracts. Analog to the network configuration, each channel has a channel configuration that defines the policies the consortium members have agreed upon. In our example, we can see that two consortiums have formed. One has the member's organizations R1 and R2 and uses channel C1, and the other consists of R2 and R3 and uses channel C2.

Channels can be seen as a logical concept, where each channel maintains one ledger, one channel configuration, and multiple smart contracts. From a physical point of view, all these channel resources are replicated over multiple peers. For an organization to participate in a channel, it has to run at least one peer, but it does not have to run one for each channel, because a peer can participate in multiple channels at once. All peers are responsible for the verification, validation, and execution of transactions. Peers that have smart contracts installed can additionally be used to create the transactions. The only way to create a transaction, and therefore modify the state of the ledger, is by using smart contracts. The example above shows a total of three peers: P1 operated by R1, P2 operated by R2, and P3 operated by R3. P1 and P2 operate the channel C1 together. The image shows that both peers have a copy of the ledger L1 and the smart contract S5. Since P2 is also operating the channel C2 together with P3, it additionally maintains a copy of L2 and a copy of the smart contract S6.

An end-user who wants to propose a transaction does not directly access the network over the peer, but uses a client application. In most cases, each organization will have its own custom application. This is also shown in the network diagram above (Figure 2.6). Organization R1 has application A1, which operates in channel C1; users of organization R2 use application A2, which operates in channels C1 and C2; organization R3 provides application A3 for its users to interact with channel C2.

The last missing part of a Fabric network is the public key infrastructure (PKI), which allows users to authenticate. Each organization operates its own certificate authority (CA1, CA2, CA3, CA4) to provide its users and nodes with certificates. These certificates can then be used by everyone in the network to verify the identity and the organizational affiliation of an actor.

2.8.2 Identity, Membership & Policies

Hyperledger Fabric policies are used at every level of the system. Fundamentally, policies declare who is allowed to access or update network resources and also provide a mechanism to enforce them. How policies work exactly is out of the scope of this thesis, but the reader is encouraged to peruse the official documentation[7] on this topic. Since policies define who has access to what, the system needs to be able to verify the identity of actors. Actors can be end-users, applications, peers, or nodes of the ordering services. Fabric uses a public key infrastructure to issue certificates, which are then used by actors to prove their identity. To connect identities to policies, Fabric uses the concept of a membership service provider (MSP). Despite its name, the MSP is just a directory structure that contains a list of permissioned identities, which can be used by organizations to decide who its admins are, and by other organizations to verify the identity of an actor.

2.8.3 Data Structure

A Hyperledger Fabric ledger consists of two components: the world state and the blockchain. The blockchain is similar to the ones used in Bitcoin and Ethereum. It is a chain of blocks linked by cryptographic hashes, where each block contains a list of transactions. Transactions describe a state transition from one state to the next. Applying all transactions contained in the blockchain in the right order results in the current state of the chain. To make it easier for applications to access the current state of the chain without having to run through the entire log and execute every transaction, the current state is stored separately as the so-called world state. Since the blockchain and the world state are used differently, they are also stored differently to optimize for the respective use case. The blockchain does not need to support querying and the only action is to append new blocks. For these reasons, Fabric uses a simple file to store the blockchain. The world state, on the other hand, is queried frequently, and these queries can be complex depending on the use case. This is why the world state is stored in a database. Currently, Fabric supports LevelDB and CouchDB as database systems for the world state. Each peer maintains their own instance of the database.

2.8.4 Consensus

Transactions are created by executing a smart contract. All transactions need to be endorsed according to the endorsing policy. Once a transaction is deemed valid and has enough endorsements, it is sent to the ordering service. If the ordering service consists only of one node, no additional consensus is necessary, because transactions are signed and validated by the channel peers. An order service composed of multiple nodes needs to make sure that all ordering nodes agree on the order and content of blocks. Hyperledger Fabric utilizes a crash fault-tolerant (CFT) consensus algorithm called Raft[OO14].

[7]https://hyperledger-fabric.readthedocs.io/en/release-2.2/network/network.html

2.9 Corda

Like Hyperledger Fabric, Corda[8] is a permissioned DLT targeted for use in businesses. At its core, Corda is free and open-source, but there is a paid plan which offers additional enterprise-grade features. For this work, the enterprise features are not necessary, because everything we want to accomplish can be done with the open-source version. Everything described in the remainder of this section is part of the open-source version. More information on the paid features can be found on the official Corda website.

2.9.1 Network

Corda takes a fundamentally different approach to the network structure than the blockchain platforms presented so far. Like the other platforms, Corda operates within a P2P network, where the peers share resources and data. However, the data is not propagated to all peers by default, but on a need-to-know basis. As a consequence, there are no peers that have a full view of the ledger by default. Each node is only aware of the states, also called facts, which it needs for its operation. It is worth mentioning that, in theory, there could be peers that have access to all the data in the ledger, but it will probably not happen in practice. Another thing is that, in a Corda network, it is not trivial to reason about who sees which information and when.

Fundamentally, the topology of a Corda network is simple, because the only required components are the peers (nodes). Peers exchange messages with each other using the Advanced Message Queuing Protocol (AMQP)[9]. The utilization of AMQP allows Corda to be resilient against network outages and node downtimes. If a node is not reachable, the message is saved in the outgoing queue and delivered once the node is online again. Operators of a Corda node can decide to offer two extra services: the network map and the notary. The network map can be seen as a distributed network registry. It stores information about all the nodes contributing to the network and is used by the nodes to discover each other. Notaries are used to provide the uniqueness consensus and are the point of finality. These terms will be explained in detail in Section 2.9.6. Once a transaction is signed by a notary, it is considered to be final and on-ledger. There is only one other special component in Corda networks, called Oracles. Oracles are optional and can be used to provide trusted off-chain information that provides additional context for transaction validation. In the context of this work, Oracles are not important; for more information on them, the reader is referred to the official Corda documentation.

[8]https://www.corda.net/
[9]https://www.amqp.org/

2.9.2 Data Structure

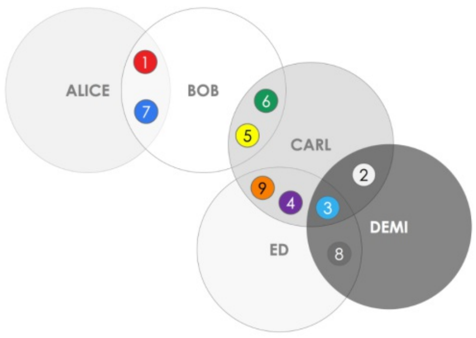

Figure 2.7: Visualization of a Corda ledger taken from the official documentation[10]

As already mentioned, Corda does not follow the conventional blockchain approach, where all information contained in the ledger is distributed to all nodes. The Corda ledger is best represented by a Venn diagram, where the intersections between nodes constitute the facts shared by these nodes. This is what is shown in Figure 2.7. The ledger contains nine facts, represented by the circled numbered 1 to 9. For example, Alice shares facts 1 and 7 with Bob. The other nodes are not aware of the existence of these facts.

On-ledger facts are represented by immutable states stored in the nodes vault. The vault is a standard SQL database, which allows for complex querying and performant access. Facts are changed by submitting a transaction that takes one or more state references as input and produces one or more output states. For each fact, there is, at any point in time, exactly one state considered to be on-ledger. Previous states are marked as historical and permanently saved for auditing purposes. By linking each state to its predecessor via its cryptographic hash value, the historical states build a sequence that represents the immutable history of a fact. In other words, Corda follows the UTXO principle. Unconsumed states can be consumed by a transaction, whereas consumed states cannot.

2.9.3 Transactions

Transactions facilitate the state transitions in Corda. They allow multiple state changes to be wrapped together in a single atomic operation. Being atomic means that either all transitions are executed or none. When transactions are created, they are called uncommitted and can be viewed as a proposal for a state transition. To become committed, all involved parties have to validate the transaction and sign it. A transaction can be viewed as committed once all needed signatures are present. This includes a signature

[10]https://docs.corda.net/docs/corda-os/4.6/key-concepts-ledger.html
[11]https://docs.corda.net/docs/corda-os/4.6/key-concepts-transactions.html

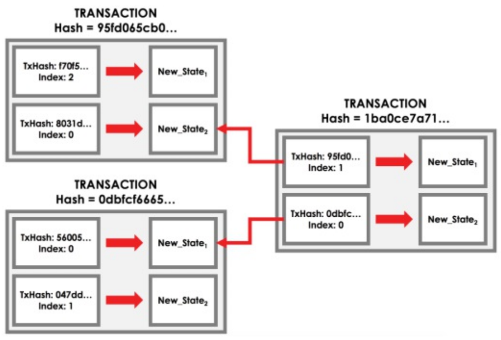

Figure 2.8: Visualization of a Corda transaction flow taken from the official documentation[11]

from a notary which provides uniqueness consensus to avoid double-spending. Notaries will be explained in more detail ins Section 2.9.6.

Corda has three types of operations, which are issuance, update, exit. These operations cover the whole lifecycle of a fact. Issuance is a transition with no input state and one output state, and is used to introduce new facts to the ledger. All changes of a fact during its lifetime are handled by the update operation. Once a fact does not need to be present on the ledger anymore, it can be removed by using the exit operation. This operation does have one input state and no output states, which effectively deletes the fact. Of course, the history of the existing fact is not affected by the exit operation. Input states of a transaction are references to an output state of a previous transaction that has not been used yet. Such state references contain the hash of the previous transaction, together with the index of the output state referenced.

Figure 2.8 shows a visual representation of a transaction flow. The transaction on the right uses state references to use one state of each transaction on the left as input states. These input states are then used to produce new output states. The referenced states are marked as spent as soon as the transaction gets committed, which means that they cannot be reused in another transaction. The transaction on the right produces two new states from the two input states, which can be used as input in a future transaction. The two transactions also have two states that have not been referenced by any transaction yet, which means they are unspent and can also be used in a future transaction. Since all state transitions in the graphic have input and output states, they are all update operations.

2.9.4 Contracts

In Ethereum and Hyperledger Fabric, the contract code defines the state transition itself. The contract code is applied to the input state and produces an output state. This code is distributed to all nodes and has to run everywhere to produce the new state and validate the transaction. Corda takes a different approach to contracts. The new state can be produced by any arbitrary code and there is no need to distribute it. As a consequence, the code which produces transactions can be proprietary. The task of the contract code in Corda is then to validate the transaction proposal. In other words, a Corda contract is a validation function that imposes constraints on the way a transaction can modify states. This validation function is distributed to all nodes on a need-to-know basis and is executed by all necessary participants to validate a transaction. Since the validation function needs to produce always the same output for the same input, there are limitations on what can be with the contract code. Corda uses a stripped-down version of the Java Virtual Machine (JVM) to guarantee deterministic contract execution. For example, the use of random number generators in the contract code would lead to different results for the same input and is therefore forbidden. The same is true for referencing external data which could change over time.

2.9.5 Flows

Since Corda does not broadcast all messages to all peers per default, ledger updates require network participants to determine which messages need to be sent to whom and in which order. Corda's Flow framework provides a programming model that facilitates these coordination operations and makes them easier to implement. A flow is a state machine that describes all steps necessary to create a valid and committed transaction. This includes signing, verifying, and transmitting proposed transactions. Flows can be short-lived and immediate, but most of the time they take many steps and messages between the parties to complete. Since a node can theoretically run thousands of flows at the same time parallelization is of major importance. The Flow framework abstracts many of the difficulties of writing asynchronous code away by utilizing lightweight threads called "fibers." Each flow runs in such a fiber until it reaches some blocking operation, like sending a message and waiting for a response. The fiber is then checkpointed, suspended, and persisted onto the disk. This makes it possible to continue flow execution even when the node crashes. Corda provides standard flows which can be called by other flows. Like in a procedural programming model where routines call subroutines to reuse code in multiple places, being able to call and reuse flows makes it possible to abstract complex interaction patterns away and make implementation easier.

2.9.6 Notaries & Consensus

Corda distinguishes between two types of consensus: the uniqueness consensus and the validity consensus. The validity consensus checks that each input and output state passes the validation function of their respective contracts and that all required signatures are present. These checks require not only looking at the transaction itself, but also verifying the chain of input state back to the creation of the state. The reason for this is best illustrated with an example. Let's assume Charlie issues 20$ and transfers it to Alice, and that only Charlie is allowed to issue money (for whatever reason). If Alice wants to buy something from Bob with these 20$, Bob has to make sure that Alice has a claim to the cash and that it was issued by Charlie. This is only possible if Bob has access to all transactions in which these 20$ were part of. In order to validate the transaction from Bob to Alice, therefore, the transaction from Charlie to Alice must also be validated. This is what is meant by transactions being propagated on a need-to-know basis, and is what makes it hard to reason about who can see transaction data after a transaction is committed.

The uniqueness consensus is what prevents double-spending. It is provided by notaries, which are pluggable services operating inside a Corda network. Notaries manage a map where the keys are transaction hashes together with an output index and the value is a transaction hash that consumed the output state referenced by the key. When a notary receives a request to sign a transaction, it looks all the input states references up in its map. If at least one of the input states is already in the map, the transaction violates the uniqueness consensus and the notary will not sign it. Otherwise, the transaction gets signed and the map updated. This is the point of finality in a Corda network. Once a transaction is signed by the notary, it is considered on-ledger and final. Each network can have multiple notaries, which all can work differently. A notary service can just be a single node, but it can also be comprised of multiple untrusting nodes. Distributed notary services can use any consensus algorithm; which one is used depends entirely on the use case. In addition to the uniqueness consensus, notaries can also provide validity consensus. This has the disadvantage that more data has to be shared with the notary, but it provides additional security against invalid transactions. Which type of notary to use is also entirely dependent on the use case.

2.10 Summary

This chapter gave an overview of blockchain technology by explaining the main characteristics of four popular implementations, all with a different feature set, different goals, and different approaches. At the core, all DLTs try to achieve the same thing, which is to maintain a distributed append-only data structure without the need of establishing trust between the involved parties. This basic concept found a wide variety of use cases. A lot of research is being conducted around the world to make blockchains part of our daily life in many different areas. Public blockchains like the presented Bitcoin and Ethereum use this to provide digital cash. The append-only nature of the underlying distributed data structure provides a detailed history of all transactions ever made without being able to link these transactions to real-world identities by default. Ethereum can be seen as an advancement over Bitcoin, because it has support for executing Turing-complete programs, called smart contracts. Smart contracts allow parties to define their transaction intentions in code and automate them. Hyperledger Fabric brings the concept of smart contracts to the enterprise world by introducing a fine-grained permission system. This permission system allows automating business processes between organizations without the need for trust or central authority. Corda has similar goals, but is interesting because it uses different concepts in a lot of areas. The most important concept is that the whole ledger is not distributed to all nodes by default, but that data is distributed on a need-to-know basis in the network. While the underlying data structure is very similar in all systems (all of them use a cryptographic hash function to link elements together), the approaches for adding new elements differ considerably. Bitcoin and the current iteration of Ethereum use the PoW consensus mechanism, which relies on solving a cryptographic puzzle to produce a new block. Ethereum will switch in its upcoming version to the PoS consensus. PoS promises to eliminate a lot of shortcomings of PoW, like high energy consumption and low transaction throughput. Permissioned blockchains like Corda and Hyperledger support multiple consensus algorithms concurrently on the same network. Different applications may have different requirements on the consensus algorithm and therefore use different approaches.

CHAPTER 3

Related Work

This chapter aims to give a comprehensive overview of the state of the art in platform-agnostic smart contracts. A systematic literature review was executed to find all relevant published articles. To guarantee that the most influential papers were found, the following steps were executed. First, the following set of search terms were defined and used to search for papers with Google Scholar:

- platform-agnostic smart contracts
- blockchain agnostic smart contracts
- smart contract programming model
- agnostic smart contract programming model
- smart contract translation
- smart contract generation

For each of these search terms, the first ten pages were searched for relevant candidate papers. Every work which could not be excluded quickly was added to an initial set of references. Next, the bibliography of each paper in the initial set was searched for possible additional relevant works. Each paper found this way also had its bibliography searched for candidates. This was repeated until no new papers were found. Once the list of candidate papers was complete, it was filtered according to the following rules:

- The work must be available in English.
- It must be freely accessible for students and employees of TU Wien.

- The work must state that it supports interacting and/or creating smart contracts on different blockchain platforms.

During the research, three major categories of platform-agnostic smart contract solutions were found. In this work they will be referenced as Model, Abstraction, and Transpiler. **Model** means here that an abstract model of a smart contract, which can be translated into implementations on different platforms, was created. **Abstraction** refers to some abstraction over the client access API. This allows the interaction with smart contracts deployed on different platforms in a transparent way. The third category is called **Transpiler**. Transpilers take smart contracts written in one language for one specific platform and perform a translation into another platform.

The remainder of this chapter is organized as follows. Section 3.1 defines the requirements a platform-agnostic smart contract platform must fulfill to be considered functionally complete. Then Section 3.2 to Section 3.10 give an overview of important works in this field. Next, Section 3.11 introduces several research topics which do not fully match the search criteria, but could be important to the field of platform-independent smart contracts in the future. Section 3.12 then gives a detailed comparison of all introduced works, along with the defined requirements. The chapter is then concluded by a summary in Section 3.13.

3.1 Requirements

The following list defines the requirements of a platform-agnostic smart contract platform or framework:

- **RQ1 - Transparent Access:** The solution makes it possible to interact with contracts defined on different platforms over the same interface.
- **RQ2 - Smart Contract Definition:** The solution provides functionality to define the data structure and the business logic of a contract once and deploy it onto different platforms.
- **RQ3 - Generality:** The smart contract definition method is general enough to support the asset management lifecycle. This includes transaction types like transfer, atomic swap, and minting.
- **RQ4 - Permission Support:** The solution works for permissioned blockchain platforms.
- **RQ5 - Maturity:** There must be an implementation available that provides the core functionality described in the corresponding paper.

- **RQ6 - Code Availability:** The code of the solution must be easily accessible and open-source.
- **RQ7 - Platform Integration Simplicity:** It should be possible to integrate new platforms easily. Here, easily refers to standard software development tasks like implementing interfaces and so on. Writing code generators is considered to be a complex task in this context.
- **RQ8 - Standard Tooling:** To support broad adoption, working with the solution does not need special tooling. All tasks can be solved by using standard tools already widely used in practice.

3.2 SCIP, SCDL, SCL - Unified Integration of Smart Contracts Through Service Orientation

Type: Abstraction

Literature: [FBD+20, FHB+19, LFB+20, FHBL19, FLB+20]

Motivation: The authors argue that blockchains and smart contracts are a new technology and lack standardization. Since different platforms provide different trade-offs, there will not be one single platform, but different ones that will coexist. This means real-world enterprise applications will have to deal with multiple platforms at the same time. Since all platforms have different APIs and data formats, this can get quite complex and work-intensive.

Approach: The whole framework consists of three main parts, the SCL, the SCDL, and the SCIP. SCL stands for Smart Contract Locator and is a concept where each contract gets assigned a URL-like and unique identifier. This identifier can then be used to uniquely identify and access smart contracts across various platforms. SCDL stands for Smart Contract Description Language and is a language following the SoA approach to describe the interface of a smart contract in a platform-independent manner. SCIP stands for Smart Contract Invocation Protocol, is an abstraction, and allows for transparent access to smart contracts deployed on different platforms. Together, these three concepts work as follows. A client application knows the SCL and SCDL of a smart contract. From the SCL, the client knows how to contact the SCIP gateway responsible for the contract in question. The SCDL, on the other hand, tells the client which transactions are available and how the corresponding message formats look. The client can now send a message to the gateway, which translates the message in a format understandable to the target platform and sends it to a node. The SCIP gateway also supports the querying of ledger data and subscribing to ledger events.

Limitations: The introduced framework does not support creating new smart contracts, but only allows for unified access to already existing contracts.

3.3 AgriBlockIoT

Type: Abstraction

Literature: [CAVG18]

Motivation: The main goal of the paper is to utilize blockchain technology for the food supply chain. A successful implementation would provide transparency, traceability, and auditability for food to everyone from the farmer to the end consumer. This would not only help to increase food quality, but also improve the safety of edible products.

Approach: The *AgriBlockIoT* framework can be divided into three layers. At the top is a REST API, which provides the capabilities of *AgriBlockIoT* to client applications. This high-level abstraction allows for easy integration into existing software systems. Below the API is the controller module. This module is responsible for translating the high-level abstractions into low-level function calls received by the lowest layer, called blockchain. The blockchain layer implements the whole business logic in the form of smart contracts. The API and controller abstractions make it possible to switch out the blockchain layer transparently. In other words, *AgriBlockIoT* supports switching blockchain platforms without the need for any adaptions in client applications. To show this, the authors implemented and deployed the proof-of-concept on Ethereum and Hyperledger Sawtooth.

Limitations: This concept only provides an abstraction for access to the data and business logic. There is no way to define the data structure and the business logic (i.e., a smart contract) in a platform-agnostic way. Each new platform requires the smart contract to be duplicated. In addition, the abstraction (i.e., the unified access to the blockchain) only works for the functionality provided by *AgriBlockIoT* and cannot be easily generalized.

3.4 Policy-Based Blockchain Agnostic Framework

Type: Abstraction

Literature: [SRS19]

Motivation: The authors recognize that there is a multitude of different blockchain platforms, each with different properties. These properties emerge from the trade-offs made in the design of the platforms. An example of a trade-off would be transaction speed vs. privacy. Naturally, different use cases have different requirements for these trade-offs. Therefore, no single platform can fulfill all use cases and many different platforms will be used for different purposes. It will be necessary for these platforms to be able to interoperate. The proposed framework should be the first step into a future of fully interoperable blockchain platforms.

Approach: The framework allows for the definition of different policies. Policy categories include cost, security, and performance. These policies are defined per blockchain and operation. Transactions are submitted transparently over a high-level API. Based on the defined policies, the ideal target platform for the transaction is determined. As an example, let us say a transaction should be executed under *MAX TIME* and cost less than *MAX COST*. The framework then determines a blockchain that can execute the transaction under these given policies. Once the adequate blockchain is selected, the respective adapter is called. The adapter then translates the transaction to the platform-specific format and sends it to a node. The last step is the execution of the transaction on the blockchain, which is done by the called smart contract. This whole concept allows for the transparent selection of a blockchain platform at runtime based on a set of policies, without the need for platform-specific knowledge.

Limitations: The work is still very much in progress and no complete implementation is available yet. Interoperability is provided with the concept of a notary. Here, the notary is the server-side component of the API and has to be trusted. Moreover, there is no support for platform-agnostic smart contracts. The framework only allows data to be stored and retrieved, but it is not possible to define business logic.

3.5 Model-Driven Approach to Smart Contract Development with Automatic Code Generation

Type: Model

Literature: [Eid20]

Motivation: The author argues that most research in the field of smart contracts is done specifically for Ethereum and Solidity. Model-driven approaches for smart contract development exists for the Ethereum platform, but the work proposes a platform-independent approach that can be used for a wide variety of platforms.

Approach: A new, high-level smart contract language is introduced. This language is based on XML and allows for the definition of data and business logic. As a short example, assume an XML document with a *Contract* element. This *Contract* element has an *ContractName* element, a *GlobalParameters* element and any number of *Function* elements as children. The *ContractName* element simply defines the name of the contract. The *GlobalParameters* is used to define fields of the contract which should be persisted. And the *Function* elements are used to declare functions. Logic inside the functions can be defined with *Condition*, *Line* and *Return* elements. A contract defined in this language is used as input for a code generator that generates the platform-specific smart contract. The output of the code generator is then deployed on the blockchain.

Limitations: A code generator has to be written for each platform that supports the proposed contract language. Since writing such a generator is complex and time-intensive, the integration of new platforms is expensive. The language is designed for public blockchains and does not support the definition of permissions. No support is provided for accessing contacts on different platforms transparently. This means that client applications need to be adapted in case the platform is switched. Contracts can be written in the same language for different platforms, but the integration in client applications is still platform-dependent. This means that client applications need a change in case the contract gets deployed on a different platform.

3.6 Unibright

Type: Model, Abstraction

Literature: [SJS$^+$18]

Motivation: Unibright is an enterprise that aims to be the first provider for blockchain-based business integration on the market. The goal is to provide high-level concepts and practical tools to integrate blockchain technology into existing enterprise systems. Unibright wants to make smart contract development easier, less error-prone, and generalized. Smart contracts developed with Unibright should not depend on a single blockchain implementation, but should be deployable on a wide range of blockchain platforms.

Approach: Smart contracts are defined as workflows. These workflows are created with the Unibright Business Workflow Designer, a low code tool for creating business workflows. These workflows are then handed over to the Unibright Contract Lifecycle Manager. This tool is responsible for creating the smart contract from the workflow, deploying it onto the blockchain, updating it if necessary, and handling general maintenance. The monitoring and querying of the installed smart contracts are done with the Unibright Explorer, which allows for unified access to contracts running on different platforms. The last component of the framework is the Unibright Connector, which is responsible for the integrations of blockchains and other IT systems into the Unibright framework.

Limitations: Unibright is a commercial product and not freely available. The code is not open-source and can therefore not be examined and reused. Unibright is specifically designed to support enterprise use cases. Contracts can only be defined visually in the form of workflows. Using Unibright involves a lot of tooling and complexity. The only description of the framework is the referenced white paper. It is not clear how far the implementation is and which blockchain platforms are supported by Unibright.

3.7 Secure Smart Contract Based on Petri Nets

Type: Model

Literature: [ZKCS20]

Motivation: Current smart contract platforms do not support designing, developing, and verifying smart contracts before they are deployed. This may lead to the deployment and use of insecure contracts in real-world applications. Such insecure contracts can lead to huge financial losses. Developing the business logic of a smart contract by designing a Petri net allows for simulation and verification before generating a smart contract. This process helps finding potential bugs early and embeds security aspects directly into the smart contract development process. In addition, developing smart contract logic visually in the form of Petri nets makes the development process easier and more approachable. The authors also mention that developing the business logic of a smart contract as an abstract model also opens up the opportunity to deploy the same logic onto multiple platforms.

Approach: This work uses Petri nets to model the business logic of a smart contract. The models are created visually, which does not require coding skills. Since fully-fledged Petri nets can be difficult to model from the ground up, the authors recommend starting the modeling of the workflow with SysML's activity diagram. The activity diagram can then be easily translated into a Petri net. Simulation can be used during the modeling process to execute a single transition or entire transactions. This helps to test the workflow logic during development and allows for fast iteration. Once the modeling process is finished, the contract enters the verification phase, executed by the proposed Verification Engine. The Verification Engine gives tips for correct and efficient workflows and can use standard Petri net validation tools. Once the developer is sure that the designed workflow is correct and secure, a code generator is used to generate the smart contract.

Limitations: Currently the authors provide only one generator which generates Solidity contracts for the Ethereum platform. Adding new platforms is complex because a code generator has to be written. There is also no possibility to model permissions at the moment, since the project is targeted at public blockchain platforms. It also does not provide a possibility for client applications to access the smart contract on different platforms in a transparent way.

3.8 Sol2JS

Type: Transpiler

Literature: [ZSJB18]

Motivation: The popularity of Solidity in the smart contract development community leads to the problem that most smart contracts are written in Solidity and can only be executed on an Ethereum network. This work tries to make all these contracts usable on other platforms by using source-to-source translation.

Approach: The authors present Sol2js, a transpiler that translates Ethereum smart contracts written in Solidity to Hyperledger Fabric contracts written in JavaScript. For this, the authors identified a mapping from Solidity concepts to Fabric concepts. So far, around 65-70 percent of Solidity keywords are supported. The team successfully translated 16 contracts, including the ERC20 token.[6]

Limitations: It only supports the translation of Ethereum contracts to Fabric contracts. It is neither possible to translate Fabric contracts to Ethereum nor are any other platforms supported. Adding new platforms is as difficult as creating Sol2js, since the work has to be repeated and a mapping from Ethereum concepts to ones of the new platform have to be found.

3.9 A Unified Programming Model for dApp Systems

Type: Model, Abstraction

Literature: [EP19]

Motivation: A real-world use case for enterprise-grade smart contracts involves many different applications written in different languages and running on different platforms. Programming the communication and translation between these applications and platforms brings a lot of overhead. Developing one system in different languages and technologies needs more know-how and is, therefore, more expensive.

Approach: The authors suggest a metaprogramming model which solves the problems stated above. The code for a complex use case involving multiple systems and platforms is written in a single language, as if everything would run on the same machine. Annotations are used to declare which parts of the code are executed on which system. The code for communication and translation between the systems is generated automatically.

Limitations: The authors did not provide any implementation details, so it is assumed that implementation has not yet started or is not far enough for publication. In theory, this model allows for the deployment of the same business logic on different blockchain platforms, as well as the transparent access to contracts on different platforms. Integrating new platforms is complex because of the complexity of code generation.

3.10 Daml

Literature: [dam]

Type: Model, Abstraction

Motivation: DAML tackles the core problems with today's smart contract development. First and foremost, DAML tries to make the development process easier and less error-prone. This includes the easier integration of blockchain applications into full solutions. Furthermore, DAML tries to solve the trade-offs between the security, privacy, and consistency of the current blockchain system. Providing interoperability and platform agnosticy are additional focus areas of the DAML project. In summary, the DAML project wants to enable the development of easy-to-use, portable smart contracts which are future-proof and therefore impose less risk for adaptors.

Approach: The core functionality of DAML is provided by the DAML smart contract language, a domain-specific language (DSL) designed for building smart contracts. It enables the developer to safely and easily describe the data structure, the permissions, the privacy properties, and the business logic of a smart contract. Each smart contract operates on top of a virtual ledger. The virtual ledger is an abstraction of the underlying distributed ledger and decouples the contract from the underlying storage and consensus. Each node has a consistent view of the virtual ledger, but the information is distributed on a need-to-know basis. Therefore, each node only receives the data it is authorized to receive. The virtual ledger is built around the concept of event sourcing, where a list of events processed in the correct order results in the current state. A target distributed ledger deployment needs to run a DAML driver on one or more nodes to use it as a data storage and consensus of a DAML application. DAML also provides a Remote Procedure Call (RPC) API that allows transparent access to the contract, regardless of the platform used in the back.

Limitations: Integrations of new platforms are complex and dependent on the DAML platform. DAML contracts need specialized tooling and use a custom language which makes adaption more difficult.

3.11 Other Mentions

Smart contracts are a very active research field, particularly because it is a relatively new technology with unique challenges. Developing a secure smart contract that behaves as expected is complicated. The most prominent example for an insecure smart contract implementation is the DAO hack[DMH17]. Considerable research is being done on how smart contract languages can be designed to make development simpler and more secure[Kne19][SKH18]. At the same time, a lot of work is being done to create tools and methodologies in the area of smart contract testing [LTHT19][LWX+19][BSS+20] and verification [DDDN19]. Other researchers are trying to find an answer to the question of who will write legally binding contracts in the future: lawyers or programmers? This is the reason for much research aiming at creating smart contracts from natural language [TYSS19][CRS+18][QTGS21]. Should any of this research be successful and lead to tools and/or languages that are adopted, these tools will likely be platform-agnostic.

3.12 Comparison

Table 3.1 shows a comparison of the solutions proposed in the literature and presented in the previous sections of this chapter. The solutions are compared by evaluating them

	3.2	3.3	3.4	3.5	3.6	3.7	3.8	3.9	3.10
RQ1 - Transparent Access	✓	✓	✓	✗	✓	✗	✗	✓	✓
RQ2 - Smart Contract Definition	✗	✗	✗	✓	✓	✓	✗	✓	✓
RQ3 - Generality	✗	✗	✗	✗	✓	✓	✗	✓	✓
RQ4 - Permission Support	✗	✗	✗	✗	✗	✗	✗	✗	✓
RQ5 - Maturity	✓	✗	✗	✓	✓	✓	✓	✗	✓
RQ6 - Code Availability	✓	✗	✗	✗	✗	✓	✓	✗	✓
RQ7 - Simple Platform Integration	✓	✓	✓	✗	✗	✗	✗	✗	✗
RQ8 - Standard Tooling	✓	✓	✓	’✓	✗	✓	✓	✓	✗

Table 3.1: Comparison of related work

- 3.2 - Unified Integration of Smart Contracts Through Service Orientation
- 3.3 - AgriBlockIoT
- 3.4 - Policy-Based Blockchain Agnostic Framework
- 3.5 - Model-Driven Approach to Smart Contract Development
- 3.6 - Unibright
- 3.7 - Secure Smart Contract Based on Petri Nets
- 3.8 - Sol2JS
- 3.9 - A Unified Programming Model for dApp Systems
- 3.10 - DAML

against the requirements defined in Section 3.1. The result of the evaluation shows that none of the solutions found in the literature fulfill all the requirements. This means that a research gap has been successfully identified. The goal of the remainder of the work is to close that gap.

3.13 Summary

This chapter described the results of the literature review in the field of platform-agnostic smart contracts. The introductory section described in detail how the literature review was executed and which keywords were used to find relevant work. Then the three main categories of solutions were identified. These categories were named Model, Abstraction, and Transpiler. Section 3.1 described the minimal set of requirements necessary for a platform-agnostic smart contract framework to be useful, easily extensible, and simple to adopt. All relevant solutions found in the literature were then introduced by declaring their type/category, explaining the motivation behind the work, giving a summary of the design and functionality, and providing a discussion of the limitations. The found solutions and the defined requirement were subsequently compared with each other. The comparison showed that no available solution fulfills all requirements and that a research gap, which will be closed by the solution presented in this work, therefore exists.

CHAPTER 4

Requirements of a Platform-Agnostic Smart Contract API

This chapter will describe the minimal set of requirements a platform-agnostic smart contract API has to fulfill to be useful and functionally complete. The core requirements were already defined in Section 3.1 and will be given in more detail in this chapter. While the goal of the definition before was to aid in the comparison of different solutions proposed in the literature, the formulation in this chapter is geared towards the implementation. For this reason, the requirements will be reformulated as user stories written from the point of view of three stakeholders. One is the Platform Integrator (PI), whose task is to add new platforms to the API. The second one is the Contract Application Programmer (CP), whose focus lies on implementing contracts. The third is called the Application Programmer (AP) and implements higher-level applications which utilize smart contracts to implement their functionality.

In addition to that, general requirements of typical smart contract applications are derived from the patterns presented in [Sen19]. This work covers the definition and analysis of the core requirements of smart contracts. The focus lies on the interaction of parties inside a community, but the presented patterns are also universally applicable.

Besides the definition of the requirements, this chapter will also describe what is not part of the scope of this work. This will cover some functional requirements, but also non-functional ones like responsiveness and security.

4.1 Interaction Patterns

Smart contracts have a wide area of application. This work will focus on a select number of interaction patterns presented in [Sen19]. These patterns were identified by analyzing several use cases in the area of community blockchains. Altogether, six patterns were identified. These patterns can be grouped into two distinct categories. Trading patterns cover typical asset management operations, whereas technology patterns cover topics like messaging and voting. This work will focus on the trading patterns of Transfer, Exchange, and Sharing. If the API presented in this work can model smart contracts which implement these three interaction patterns, the API is considered general enough to support a wide range of use cases.

4.1.1 Transfer

In [Sen19], the Transfer pattern is limited to fungible objects. In this work, this pattern is extended to also include non-fungible objects. Described in short, a Transfer is a unidirectional change of ownership of some object. This object can be fungible, like coins, or non-fungible, like a piece of art. Here, "unidirectional" means that the transfer is only in one direction. In other words, the sender does not get anything back for transferring the object, at least not atomically in the same transaction.

4.1.2 Exchange

In contrast to a Transfer, an Exchange is a bidirectional transfer of an object. It can be viewed as two transfers packed together in one atomic transaction. One transfer from participant 1 to participant 2 and one transfer the other way around. If objects of both Transfers are non-fungible, this operation can also be called a swap.

4.1.3 Sharing

Sharing describes the joint usage of a non-fungible object by at least two participants. In contrast to Exchange and Transfer, there is no change in ownership of the object in question. The borrower simply gets granted access rights to the object. These can be valid for a predefined period or until revocation.

4.2 User Stories

This section will describe the requirements of a platform-agnostic smart contract API from the point of view of the stakeholders. The API can be viewed from three perspectives. One comes from the distributed ledger to the API. This is the direction in which a developer who wants to integrate a new platform would view the API. In the rest of this work, this role will be called *Platform Integrator*. The second direction comes from an application that wants to utilize custom-developed smart contracts. This role will be called *Application Programmer*. The third viewpoint is that of the *Contract Application Programmer* who is responsible for implementing smart contracts. The user stories introduced in the following sections are derived from the requirements listed in chapter 3 and the interaction patterns explained in section 4.1.

4.2.1 Platform Integrator User Stories

As a *Platform Integrator...*

- (PI1) I want to be able to integrate new platforms by implementing an API, instead of writing code generators.
- (PI2) I want to be able to provide a high-level API to the permission system of the platform I want to integrate.
- (PI3) I want to use standard tooling and languages for implementing the presented API on the application side. On the platform side, the tooling provided by the platform should be used.

4.2.2 Contract Application Programmer User Stories

As a *Contract Application Programmer...*

- (CP1) I want to create an asset type with arbitrary fields.
- (CP2) I want to store collections of defined asset types on the ledger.
- (CP3) I want to add new assets to collections on the ledger.
- (CP4) I want define transfer transactions as described in Section 4.1.1.
- (CP5) I want to define exchange transactions as described in Section 4.1.2.
- (CP5) I want to define share transactions as described in Section 4.1.3.
- (CP6) I want to use a reference implementation for contract development to avoid having to deploy a whole ledger network for development.

4.2.3 Application Programmer User Stories

As an *Application Programmer...*

- (AP1) I want to execute a transaction defined by smart contracts implemented with the Ledger Access API.
- (AP2) I want to configure which distributed ledger platform should be used in the backend.
- (AP3) I want to use a reference implementation for application development to avoid having to deploy a whole ledger network for development.

4.3 Outside the Scope

There are several topics that remain outside the scope of this thesis. The focus of this work lies in showing that it is possible to develop an API that allows smart contracts to be written once and run on different platforms. The goal is to show how such an API could look and not to deliver a fully functioning product which can already be utilized by end-users. The result will be a prototype that covers the most interesting and difficult parts of such an API and leaves some implementation details open. Whenever such details are omitted by the implementation, this will be pointed out in the thesis and an explanation on how this would function in a fully-featured application will be given. It is also worth noting that this work does not focus on the security aspects of the contracts developed utilizing the API. Showing that contracts executed with the API still possess all the same security properties as the underlying platforms is a topic for future work. This thesis will also not cover the performance aspects. It is not likely that implementing contracts using the Ledger Access API has a performance impact compared to using native contracts, but showing this is also a topic for future work. Topics related to future work will be explained in more detail in Chapter 12.

CHAPTER 5

Ledger Access API

This chapter introduces the *Ledger Access (LA) API*, a collection of classes and interfaces which enable programmers to write smart contracts once and deploy them onto different distributed ledger platforms. The API can be logically divided into two parts. One represents a general interface that allows for the reading and writing of data from/to a distributed ledger, called *Ledger Platform Interface* or *LPI*. The other one is called *Smart Contract Interface* or *SCI* and is responsible for providing the components needed for smart contract development. This includes a way to define the contract's data structure, the state transition logic, consensus rules, and helpers that make the development process faster and easier. The remainder of this chapter will explain the LA API in detail and is organized as follows. Section 5.1 will give a general overview of the whole API architecture and a high-level description of the functionality. Then, Section 5.2 will introduce the SCI models that can be used to implement smart contract applications. After that, Section 5.3 will describe the design of the platform facing interface. In Section 5.4, some limitations and their consequences of the *LA API* will be discussed. The chapter will be concluded with a summary in Section 5.5.

5.1 Overview

Figure 5.1 shows the components of the *LA API* and how they interact with surrounding modules. As already mentioned, the *LA API* can be split into two parts, the LPI and the SCI. In the diagram, as well as in the remainder of this work, an implementation of the LPI is called a *Platform Adapter*, whereas the implementation of the SCI is called a *Contract Application*. Applications which use the LPI and SCI to execute transactions on a ledger are called *Client Applications*. Transactions are provided by the *Contract Application* and submitted over a *Platform Adapter* by a *Client Application*. From the view of a distributed ledger network, the *Platform Adapter* acts as a client, as it invokes

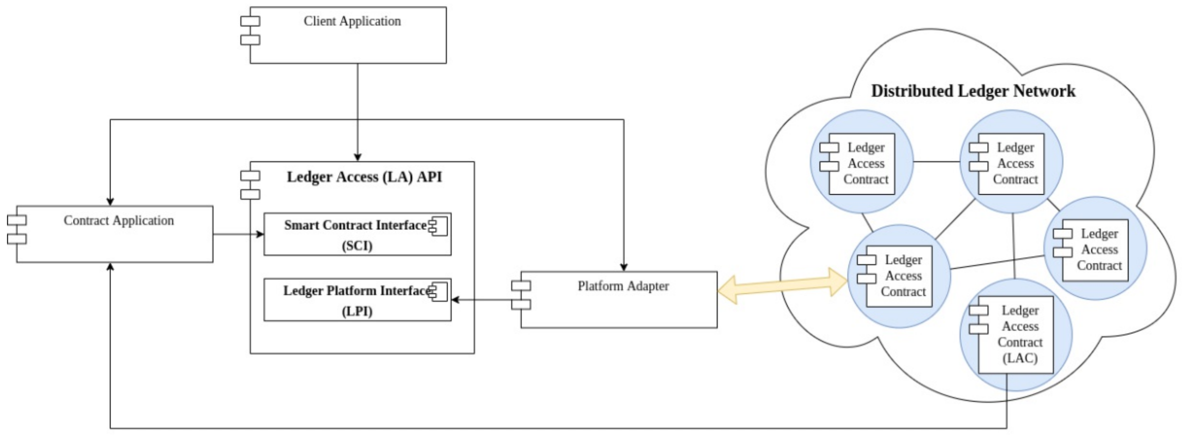

Figure 5.1: Ledger Access API

transactions of smart contracts deployed on the network. In the case of a *Platform Adapter*, the invoked transactions are all part of the *Ledger Access Contract (LAC)*. The *LAC* is a generic contract that allows the execution of all transactions to be defined by *Contract Application*s. This means that, like the *Platform Adapter*, the *LAC* must only be implemented once for each platform.

Let's assume a use case for some existing application arises, where a smart contract on a distributed ledger is the best way to solve it. For reasons explained earlier in this work, it is decided that the smart contract should be implemented with the *LA API*. Luckily, a *Platform Adapter* including the *LAC* is already implemented and deployed on the network. In this case, step one is to implement the desired smart contract functionality as a *Contract Application*. The data structure, the state transition logic, and the consensus rules have to be defined following the models provided by the *SCI*. Section 5.2 will explain the *SCI*-related steps in more detail. The resulting *Contract Application* is then added as a dependency to the *LAC* and the *Client Application*. The *LAC* needs to know about all *Contract Applications* in order to be able to validate them. How the validation is done and what is validated will be detailed in Section 5.2. The *Client Application* additionally adds dependencies for the *LA API* and the desired *Platform Adapter*. These steps are all necessary for a *Client Application* to be able to invoke transactions defined by the *Contract Application* on the target ledger using a *Platform Adapter*.

When the *Client Application* invokes a transaction, it does so by calling the respective function of the *LPI*. Depending on which target platform is configured, the *LPI* call will be received by the corresponding *Platform Adapter*. The *Platform Adapter* translates from the *LA API* domain to the platform domain and communicates the transaction details to the *LAC* utilizing whatever protocol the target platform provides. The *LAC* then checks the transaction and executes it when all the validations have passed. A valid transaction is a transaction with a valid state transition and all necessary signatures. How valid state transitions and the set of necessary signatures are defined and validated will be explained in detail in the remainder of this chapter. The resulting state after the execution of a transaction can then be retrieved by the *Client Application* using the *LPI*.

5.2 Smart Contract Interface

This section describes the architecture of the proposed smart contract API. This is the API programmers can use to create platform-agnostic smart contracts. The design is best explained by breaking it down into the main components, also called models here. The general concept of the API can be described as follows:

Each smart contract has to have some state associated with it. The structure of this state is defined by deriving a class provided by the *SCI*. Several fields are required and therefore have to be implemented, but other than that, there are no limitations to the content and structure of the state. An arbitrary amount of arbitrary type fields can be added to state classes. All these properties will then be part of the *Contract Application*'s state and persisted on the ledger.

State changes are modeled with a structure called transactions. A transaction object consists of three main elements: an old state, a new state, and a command. The old state is the starting point of the transaction. Everything contained in the old state must already be persisted on the ledger. To issue transactions, the old state is empty. The new state contains the old state with all the changes the transaction should make to the state already applied. The command has two purposes. One is to declare the required signers of the transaction and the second is to declare the type of the transaction. The transaction type is important for the transaction validation, whereas the declaration of the signers is important for consensus. Transaction validation is done by a validator. While transaction generation can happen outside of the ledger (but from trusted code), transaction validation has to be done on-ledger to ensure that the defined rules for state transitions are enforced by the blockchain network. Each *Contract Application* provides a validator that declares the state transition rules. Based on the command of the transaction, the *LAC* decides which validator will be loaded and used.

The validator only decides if the state changes of the transaction are valid, but not if the necessary parties have agreed to execute the transaction. For that, the *LPI* (see Section 5.3) provides unified access to on-chain identities. These identities are used to declare who has to sign a transaction. A required signer is added to a transaction by adding its identity to the list of signers of the transaction command.

To reach consensus, transactions are sent to the required participants to gather their approval in the form of signatures. This process is either handled by the platform itself or implemented by the *Platform Adapter*. Either way, the collection of signatures is the responsibility of the *LPI*. Since the transactions are hashed and the hash is also signed, all participants can be sure that only transactions which have been agreed upon by the necessary parties are executed. Any changes to the transaction after signing are easily detectable and lead to the rejection of the transaction.

From the description of the *SCI* above, the functionality can be split into three main components, also referred to as models. The remainder of this section will describe these models in detail. First, Section 5.2.1 will cover the definition and usage of the state model. After that, Section 5.2.2 will show how transactions are defined and how validators can be implemented by describing the transaction model. Next, Section 5.2.3 details the consensus model, a way to define the set of required signatures for each transaction. Last but not least, Section 5.2.4 introduces a helper class which facilitates the implementation of *Client Applications* and *LAC*s.

5.2.1 State Model

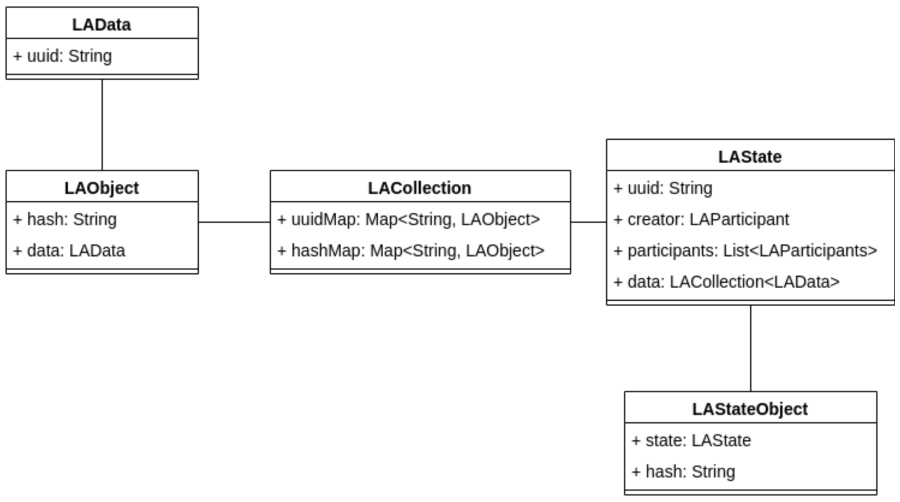

Figure 5.2: State Model

The state model of the *LA API* is used to define the data managed by a *Contract Application*. Figure 5.2 shows the class structure of the state model as defined by the API. The model is designed with two main goals in mind. One is to make it as easy as possible for a smart contract developer to define the data model for their specific use case. The other is to provide base functionality used by the transaction and consensus models to enforce permissioned blockchain and smart contract properties like immutability, access control, and decentralized trust. To reach these goals, certain compromises leading to limitations in terms of flexibility had to be made. One limitation of the model is that it is not general enough to support all possible smart contract use cases. Use cases not concerned with asset management may not be supported. The goal of the model is specifically designed to manage a collection of assets. Managing just a single object is not directly supported, but possible. This fact should not exclude too many use cases, but its effects will be discussed in more detail in Section 5.4.

A developer who wants to define the state associated with a *Contract Application* creates a class that inherits from *LAData*. The only requirement of a data object is to have a

UUID associated with it to make it uniquely identifiable. Everything else is up to the developer. Custom fields and properties can be defined without any limitations. *LAData* objects get wrapped together with the hash of their serialized string representation into an *LAObject*. This is necessary for the transaction layer to determine if an *LAData* object has changed or not. Objects of the *LAObject* type are then used in the *LACollection* class to refer to data objects. *LACollection* is a support class used to make comparison operations easier. These operations are used heavily in the transaction model to find the differences between two states (i.e., the old state and the new state). This topic will be discussed in more detail in Section 5.2.2. The *LAState* class adds information from the permission layer, such as the creator and the participants, to the model. This gets re-wrapped together with its hash into the *LAStateObject*, which represents the top-level object of the state model hierarchy. This is the class that will be primarily referenced in the other models.

5.2.2 Transaction Model

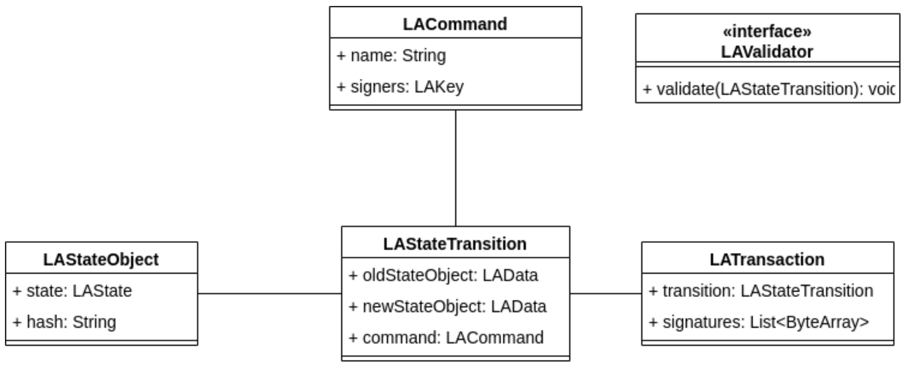

Figure 5.3: Transaction Model

While the state model defines the data managed by a *Contract Application*, the transaction model defines how the data can be changed. In theory, the model can express every possible transition from one state to another. This is not useful in the context of smart contracts, because the whole goal is to restrict the possible state changes to a set agreed upon by all participants. This restriction is enforced by defining all valid types of transactions, giving them a name and a set of rules. The name provides information about the intent of a transaction type and is also referred to as the command of a transition.

The transaction model can be seen as a simple state machine, where the state is represented by *LAStateObjects* and transitions are declared as *LAStateTransitions*. An *LAStateTransition* object consists of two *LAStateObjects* which represent the old and the new state and a command in the form of an *LACommand* object. Enforcement of the transition rules is done by the *validate* function of the *LAValidator* interface. This function takes an *LAStateTransition* as a parameter. By accessing the command of the transition, the function can determine which rules have to be enforced. If the logic in the validator function finds an erroneous transaction, an exception is thrown; otherwise, the function terminates successfully. There is no need for a return value. An example of such a transition rule is that when a transfer transaction is executed, only the owner of the object is allowed to change. All other properties have to stay the same. The key concept here is that the validation function is declared by the smart contract developer, agreed upon by all participants, and executed on the blockchain. This is important for all participants to have trust that no invalid transactions are executed.

The signer field of the *LACommand* class, as well as the signature field of the *LATransaction* class, are necessary to be able to define the consensus model of the smart contract. This will be discussed in more detail in Section 5.2.3.

The only part of the transaction model which needs to be provided by a smart contract developer is an implementation of the *LAValidator* interface. The whole application logic of the smart contract is embedded in the validation function. This function defines the set of valid transitions and therefore the business logic of the *Contract Application*.

5.2.3 Consensus Model

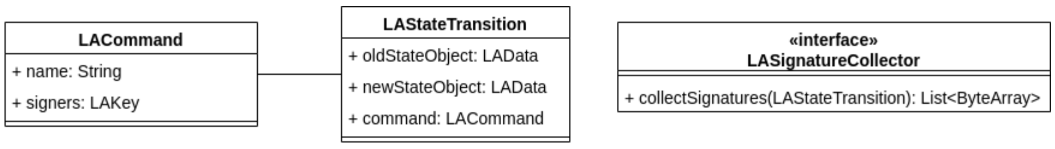

Figure 5.4: Consensus Model

The consensus model describes by whom a transaction has to be signed in order for it to be considered valid. This is done on a transaction instance level and not on a transaction type level. The reason for this is that the exact signers are not known at implementation time. When transferring an asset, for example, at development, it is only known that the current owner has to sign. Who the current owner is can only be determined at runtime.

Figure 5.4 shows the classes of the *SCI* that are involved in the consensus model. These are mainly classes from the transaction model which were already introduced in Section 5.2.2. The party having to sign a transaction is determined by the *signers* field of the *LACommand* contained in the *LAStateTransition*. The *LAStateTransition* can be seen as a proposal for a state change. A transaction following the consensus rules (i.e., one that is considered agreed upon by the relevant parties) is an *LAStateTransaction* object with the signatures of the participants declared by the contained *LAStateTransition* object. This means whoever generates the *LAStateTransition* and, therefore, the transaction, has control over who has to sign the transaction. For this reason it is very important that transactions are only generated from trusted and agreed upon code. Otherwise, everyone could create arbitrary transactions and sign themselves. This is a limitation which will be further discussed in Section 5.4 and in Chapter 12. The fact that the signers of a transaction are declared dynamically during transaction generation also has advantages. All information available at the time of generation can be used to create arbitrary business logic to determine who has to sign. This means that, in theory, there is no limitation on the consensus rules supported by this model.

The actual gathering and verification of the signatures is the responsibility of the *LPI* and is described in detail in Section 5.3.

5.2.4 Contract Configuration

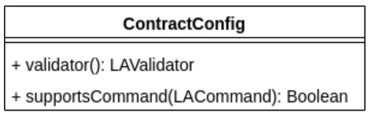
ContractConfig
+ validator(): LAValidator
+ supportsCommand(LACommand): Boolean

Figure 5.5: Contract Config

Figure 5.4 shows the last missing part to make the *SCI* into the *LAContractConfig* interface. Each *Contract Application* has to provide an implementation of this class. The diagram shows only the essential methods. In reality, the interface provides a few more methods that make the development easier, but are not crucial for the core functionality of the *SCI*.

The *LAContractConfig* class is used to register new *Contract Application*s inside the *LAC* and *Client Application*s. If an implementation is found, it is added to a list of supported *Contract Application*s. How these implementations are found is an implementation detail and will be explored in Chapter 5. Here it is just assumed that such a list exists. When an *LATransaction* is submitted to the *LAC*, the first step is to retrieve the contained *LACommand* from it. Using this command object, each registered config can be checked to see if the command is supported. Since each command is supported by exactly one config, this method allows the config of the *Contract Application*, which declares the given command, to be retrieved. This config can then implement all sorts of application-specific logic, but most importantly it can be used to retrieve an instance of the *LAValidator*.

In other words, the *LAContractConfig* allows for the retrieval of the correct *LAValidator* for a given transaction type.

5.3 Ledger Platform Interface

The goal of the *Ledger Platform Interface (LPI)* is to provide unified access to the distributed ledger. "Unified" meaning in this context that the access to the ledger works the same, regardless of which platform is used in the backend. Figure 5.6 shows the *LPI* and its surrounding components. Permissioned blockchain platforms have two components the external world needs access to, and they are already contained in the

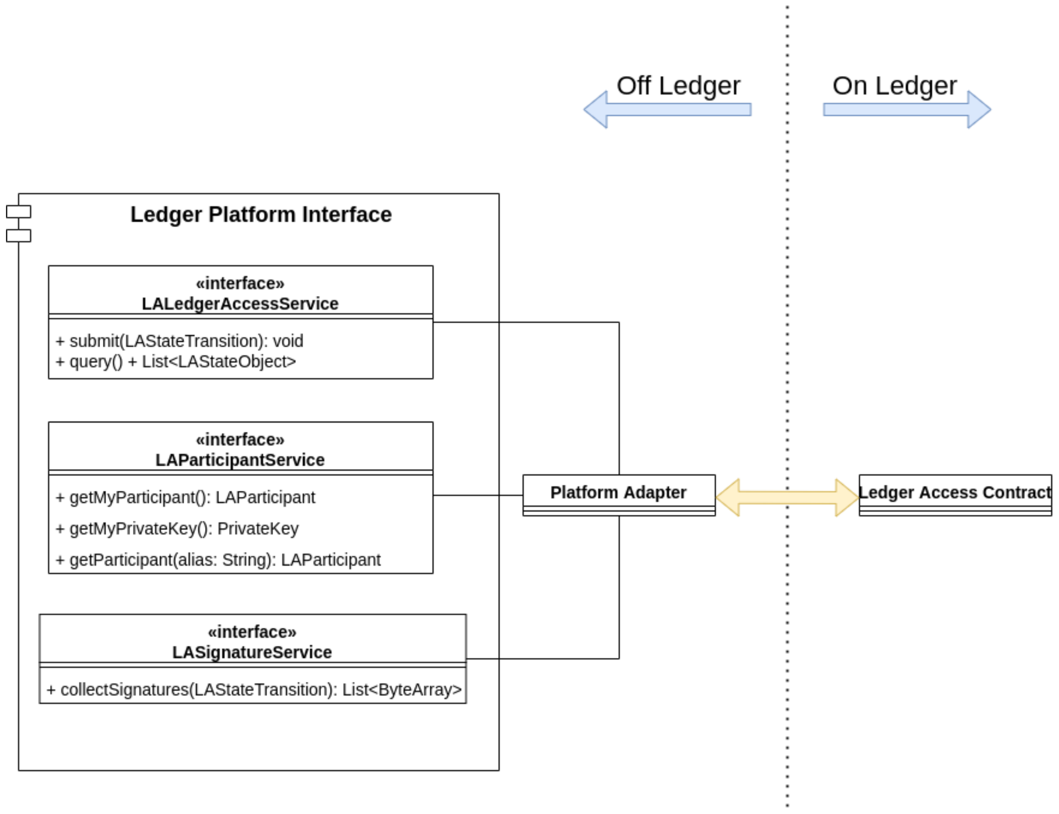
Off Ledger
On Ledger
Ledger Platform Interface
«interface»
LALedgerAccessService
+ submit(LAStateTransition): void
+ query() + List<LAStateObject>
«interface»
LAParticipantService
+ getMyParticipant(): LAParticipant
+ getMyPrivateKey(): PrivateKey
+ getParticipant(alias: String): LAParticipant
«interface»
LASignatureService
+ collectSignatures(LAStateTransition): List<ByteArray>
Platform Adapter
Ledger Access Contract

Figure 5.6: Ledger Platform Interface

name. The first one is the blockchain itself, as contracts need read and write access to the ledger in order to persist state in a trusted way. Since the *LA API* targets permissioned blockchains, access to the identities is also required. The third task of the *LPI* is to collect signatures and is optional. If the target platform supports signature collection, the functionality provided by the platform should be used. When its not supported, the *LPI* can be used to implement it. The functionality described above is provided by implementations of the *LALedgerAccessService*, the *LAParticipantService*, and the *LASignatureService*, respectively. An implementation of the LPI is called a *Platform Adapter*. Each supported platform has its own *Platform Adapter*, and integrating a new platform requires the creation of a *Platform Adapter*. All the components discussed until now are off-ledger, which means that they do not need to run on the nodes of the distributed ledger, but can be executed anywhere. In real-world applications, these components will probably be parts of the client applications. These client applications reference the *LPI* at development time and do not need to know which platform will be used at runtime. At runtime, the *Platform Adapter* for the target platform is provided as a runtime dependency and handles the calls to the *LPI*. A fully functional platform integration also needs an on-chain component, called *Ledger Access Contract (LAC)*, the purpose of which is to implement the transactions required by the *LPI*. Precisely which transactions are needed is (mostly) platform-dependent, but the *submit* method of the *LALedgerAccessService* is an example of a transaction that needs an on-ledger counterpart on most platforms. More details on the implementation of the *LAC* will be given in Chapter 8.

From the view of a developer who wants to integrate a new platform into the *LA API*, the following tasks need to be completed. First, a *Platform Adapter*, which contains implementations of the *LALedgerAccessService*, the *LAParticipantService*, and (if needed) the *LASignatureService*, has to be created. The main purpose of these services is to handle the communication with the ledger and map the requests/responses to/from the *Ledger Access* domain to/from the domain of the used platform. Implementing these off-chain services will reveal which functionality is provided by the client API of the platform and what needs to be implemented by an on-chain contract. The missing or LA API-specific functionality then has to be provided in the form of the *LAC*. The *LAC* is implemented utilizing the contract API of the target platform. It should be noted that the *LAC* is a general-purpose contract. "General-purpose" meaning here that it can handle all contracts implemented using the *SCI*. Multiple smart contract instances containing different logic and serving different purposes can all target the same *LAC* instance.

Although most platforms support writing smart contracts in multiple programming languages, not all languages are supported by all platforms. This fact imposes some limitations on how DLT platforms can be integrated into the Ledger Access API. This topic will be discussed in more detail in Section 5.4.

5.4 Limitations

Developing smart contracts utilizing the *LA API* has several limitations which will be discussed in detail in this section. The order in which the limitations are named is arbitrary and has nothing to do with their severity or impact.

First, the state model has a limitation. It is specifically designed to support asset management use cases, where a collection of assets should be managed. Use cases managing single assets are supported, but the state model provides numerous features geared towards more complex use cases and makes the implementation of these simpler ones more difficult than it has to be. If many use cases which need to be able to manage single assets arise, it is recommended to create a second state model geared towards this class of applications. The transaction and consensus models could be used as is. Moreover, since this work focuses on asset management-based use cases, other types of applications may or may not be supported by the state model. All use cases which target the management of the lifecycle of a collection of assets are supported without any limitations.

The second limitation does not affect the smart contract development itself, but limits the selection of possible distributed ledger platforms used in the backend. The permission layer of the *LPI* expects that some form of asymmetric key encryption is used to identify participants in the network. This is necessary to map the identities from the ledger platform to the classes used by the *LA API*. Furthermore, either the underlying platform supports the collecting of specific signatures to approve transactions, or it must be possible to acquire the private key of the current participant. Here, the current participant refers to the identity the *LA API* is using to communicate with the blockchain. Otherwise, it is not possible to collect the necessary signatures and consensus cannot be reached.

Another limitation for the types of platforms which can be integrated comes from the languages supported for writing smart contracts. As described in Section 5.3, the platform adapter communicates with a custom smart contract running on the ledger itself. Transactions submitted to this contract must be validated by an implementation of the *LAValidator* interface. This validator is provided by an implementation of the *SCI*, i.e., the *Contract Application*. Assuming that the validator will be implemented in just one language, only those ledger platforms that support writing contracts in the language the validator uses can be integrated.

5.5 Summary

This chapter introduced the design of the *LA API*, a programming model which allows for platform-independent smart contract development. The API can be divided into two major interfaces, the *LPI* and the *SCI*. The *LPI* abstracts the communication with the target platform and allows data to be read and written on different platforms over the same interface. The *SCI* is designed to allow developers to define state structure, transactions including validators, and consensus rules of smart contract applications. For this purpose, the interface provides three models, which handle these different aspects of the smart contract. First, the state model was introduced. Its purpose is to give the developer an easy and flexible way to define the data that should be managed by the smart contract. Secondly, the transaction model was proposed to give the developer a way to define the logic that governs the change of the previously modeled state. The most important function of a blockchain is to make sure that only transactions agreed upon by the necessary participants are executed and immortalized on the blockchain. That functionality is handled by the consensus model. Transactions generated using the models described above can be submitted over the *LPI* to the *LAC*. The *LAC* performs the required validations and, if successful, the proposed changes are applied to the state. The chapter was concluded by a discussion of the limitations the *LA API* imposes. Potential target platforms need to use some form of asymmetric encryption to identify participants. It is also preferred that target platforms support writing contracts in Java. In terms of flexibility of the smart contract programming model, the state model is designed for use cases that manage a collection of assets. Other use cases may be supported, but are not the topic of this thesis.

CHAPTER 6

Use Case - Asset Management

This chapter introduces a use case with the goal of covering all requirements of the *Contract Application Programmer* and *Application Programmer* roles defined in Chapter 4. The requirements of a *Platform Integrator* are addressed by the implementation of *Platform Adapter*s in Chapter 8. A working implementation of this use case will show that the presented API fulfills all the requirements of a platform-independent smart contract framework that were initially gathered. The definition of the use case will be aligned with the structure of the *SCI* defined in 5.2. In theory, use cases can be defined in any arbitrary way, but defining them in terms of the *SCI* provides the advantage of a direct mapping from design to implementation. In this way, the involved steps and the general idea are easier to follow and comprehend. It is recommended that developers who want to implement a smart contract with the *LA API* follow the development process presented here and define their use case in a similar manner. This chapter will only cover the design and the requirements of the use case; the implementation will be given in Chapter 9. The remainder of this chapter is organized as follows. Section 6.1 explains the general idea behind the use case and describes its core functionality. Next, the three models of the *SCI*, namely the state model (6.2), the transaction model (6.3, and the consensus model 6.4, are defined. The chapter is concluded with a summary in Section (6.5).

6.1 Description

The main purpose of the use case is to show that all major transaction types of an asset lifecycle can be implemented with the *LA API*. This includes the issue, transfer, exchange, lending, and updating of properties. These essential transaction types were examined in Chapter 4. The exact type of asset used in this example is not important and could theoretically be anything. For this work, a car manufacturer who wants to manage their cars over their lifetime was chosen as an illustrative example. The requirements of the car manufacturer are as follows. They want to manage a collection of cars containing all cars manufactured after a certain date. Cars manufactured before the smart contract was in production are not managed by it. The manufacturer only sells directly to the customers and does not use third-party dealerships. It is important that the cars can be transferred and exchanged without any limitations. The keyless access system of the cars queries the blockchain and allows people saved as drivers to drive the car. In other words, if the owner of the car wants to share it with someone, this person needs to register his/her identity on the blockchain and has to be added to the list of drivers of this car. To summarize, this use case covers the management of a collection of cars. This collection will be called "fleet" for the remainder of this work. Cars contained in this fleet can be transferred and exchanged. In addition, the owner can share his/her car with someone by adding his/her identity to a list of drivers.

6.2 State Model

The state model describes the data managed by the smart contract. There are a few requirements to the state model coming mainly from the transaction types which should be supported. There should be some way to uniquely identify a car. Moreover, cars of the same type and with the same owner should have different hash values. To make transfers and exchanges possible, there must be some concept of ownership. Because of the sharing requirement, ownership is not enough to determine who is allowed to drive a car. Therefore, the state must also model the allowed drivers. Putting this all together, a model of a car for the given use case needs the following properties:

- **UUID** - used to uniquely identify a car, changes the hash value even if all other values remain the same
- **Brand** - exemplary property with no special purpose
- **Model** - exemplary property with no special purpose
- **Owner** - the owner of the car
- **Drivers** - people/participants who are allowed to drive the car

It should be noted that the *brand* and *model* properties are not strictly necessary for the use case, but facilitate the understandability and clarity of the example. Since the goal of the use case is to manage multiple cars and not just a single one, the state model needs to provide some kind of container object which contains a collection of cars. Like the cars (i.e., instances of the car model), it must be possible to uniquely identify a collection of cars (i.e., a fleet). A fleet also has to have a creator, who is also the owner, to determine who is allowed to add new cars to the fleet. In this example, the creator of the fleet is a car manufacturer. A list of participants is necessary to determine who is part of the contract. Of course, the fleet must also include the list of the cars themselves. Summing these requirements up, the *Fleet* model must have the following properties:

- **UUID** - used to uniquely identify a fleet
- **Creator** - creator and owner of the contract
- **Data** - collection of all the cars in the contract
- **Participants** - list of all car owners, drivers, and the creator of the fleet

6.3 Transaction Model

The transaction model represents the business logic of the smart contract. Each transaction type derived from the requirements of the use case has to be declared here in the form of a command. As described in Section 5.2.2, each transaction is composed of a state transition and the signatures of said transition. Each transition is modeled by the previous state, the next state, and the command. The validator takes such a transition, determines the intent of the transition by retrieving the command, and enforces certain rules for the transition at hand. The remainder of this section will describe the rules the validator needs to enforce for each transaction type. The outcome of the transaction model is the validator and the commands.

6.3.1 Create Fleet Transaction

The purpose of the create fleet transaction function is to create a new empty fleet. In our case, this transaction will be executed by the car manufacturer. The rules for this transaction are as follows:

- The previous state has to be empty.
- The data has to be empty, i.e., the fleet should not contain cars in the beginning.
- There should only be one participant in the contract, who has to be the creator.
- The fleet must have a UUID associated with it.

6.3.2 Create Car Transaction

The create car transaction adds new cars to a fleet. Like the fleet creation transaction, the create car transaction is executed by the owner or the fleet, i.e., the manufacturer. The following rules need to be followed for a create car transaction to be valid:

- The transition has to reference a previous state.
- The UUID of the old state and the new state must match.
- Exactly one new car must be added to the fleet.
- No existing cars should be changed in any way.
- No cars should be deleted from the fleet.

6.3.3 Transfer Transaction

The transfer transaction is used whenever a car is sold. This transaction is executed by the current owner of the car. The rules for the transition are as follows:

- The transition has to reference a previous state.
- The UUID of the old state and the new state must match.
- No new car should be added to the fleet.
- No cars should be deleted from the fleet.
- Exactly one car is allowed to change.
- The only property of the car allowed to change is the owner.

6.3.4 Exchange Transaction

The exchange transaction is used to swap the owners of two cars. It represents the common use case of an atomic swap, where either the owners of both cars are changed or nothing happens. This prevents one person from ending up with the ownership of both cars. The following rules need to be enforced for an exchange transition to be valid:

- The transition has to reference a previous state.
- The UUID of the old state and the new state must match.
- No new car should be added to the fleet.
- No cars should be deleted from the fleet.
- Exactly two cars should change.
- The current owner of Car1 should be the previous owner of Car2.
- The current owner of Car2 should be the previous owner of Car1.
- No other properties of both cars are allowed to change.

6.3.5 Share and Revoke Share Transaction

Implementing the sharing functionality requires two transactions. One is the sharing transaction itself, where a car is shared by the owner with another participant. The second one is initiated by the owner and revokes the driving right of a participant. This transaction can be seen as the reverse of the first one and is called the revoke share transaction. Both transactions need to adhere to the following rules:

- The transition has to reference a previous state.
- The UUID of the old state and the new state must match.
- No new car should be added to the fleet.
- No cars should be deleted from the fleet.
- Exactly one car is allowed to change.
- The only property allowed to change is the list of drivers.

The way in which the transactions differ is how the list of drivers is allowed to change. For the share transition, the following rule must be adhered to:

- There must be exactly one driver *added* to the list of drivers.

Whereas the opposite must be enforced for the revoke share transaction:

- There must be exactly one driver *removed* from the list of drivers.

6.4 Consensus Model

The consensus model determines who needs to agree to a transaction for it to be allowed to be executed. A command of a specific transition not only declares its intention, but also has a list of necessary signers. In other words, the participants who need to sign are different for every transaction, but depend on the role they currently have in the contract. The following is a list of which roles need to sign which transactions:

- **Create Fleet Transaction:** Only the creator has to sign. It can be executed by anyone.
- **Create Car Transaction:** Needs to be signed by the creator of the fleet in which the car should be created.

- **Transfer Car Transaction:** Needs to be signed by the seller, who has to be the owner at the point of sale. In this case, the buyer does not have to sign the transaction, because it is assumed here that receiving a car unwanted has no negative effects. In other cases, it might be necessary for the seller and buyer to sign a transfer transaction.
- **Exchange Car Transaction:** Needs to be signed by both owners.
- **Sharing Transactions:** Needs to be signed by the owner of the car. The same reasoning as for the transfer transaction; in other cases, it might be necessary for the participant who receives the shared object to sign.

6.5 Summary

This chapter gave a comprehensive description of a possible smart contract use case. The described fleet management use case is designed to cover the most common phases of the asset management lifecycle. This covers transactions like creation, transfer, exchange, and sharing. The use case was described following the models introduced in Chapter 5. This once again showed in more detail the purpose of each model and how they all interact. With this detailed description, it is also easily possible to adapt the use case for assets other than cars. In later chapters of this work, this use case will be used to show the versatility of the proposed smart contract programming model, as well as the correctness of the implementation on the target platforms.

CHAPTER 7

Technology Stack

This chapter covers the languages, tools, frameworks, and libraries used for the implementation part of this thesis. This includes the implementation of the *LA API*, the *Platform Adapters*, the *Contract Application*, and the *Demo Client*. The following chapters will cover in detail how these components are implemented with the technologies listed here.

7.1 Language

The goal was to select a programming language that can be used to implement all components necessary for a fully functional prototype. This includes the *LA API* itself, the *Platform Adapters*, the *Contract Applications*, and the *Demo Client*. Based on this requirement, the selected platforms (*Fabric* and *Corda*), and the previous experience of the development team, the *Kotlin*[1] programming language was selected. *Kotlin* is a programming language developed and provided for free by *JetBrains*, a software company known for their IDEs and other tools widely used in professional software development. It is a multi-purpose and multi-platform language, with the most popular usages being server-side development and *Android* app development. *Kotlin* is officially recommended by Google for *Android* development. Besides that, it is also possible to compile *Kotlin* code to *JavaScript* and therefore it can be used to develop web frontends.What makes it interesting for our use case is its support for the *Spring Boot* framework, which enables *Kotlin* programs to interact with *Java* code without limitations, and the code to be run on every standard *JVM* just like normal *Java* code. This gives us access to the vast *Java* ecosystem with its high-quality libraries, while also offering state-of-the-art language features provided by *Kotlin*.

[1]https://kotlinlang.org/

7.2 Build Tool

The language selection determines the possible options for build tools. In the JVM world there are only two options. These are *Apache Maven*[2] and *Gradle Build Tool*[3]. Both build tools support the declaration of dependencies, the extension of plugins, and the whole process of building and delivering software. This includes compiling, testing, packaging, and publishing. The main distinction between the two tools lies in the way the build process is described. *Maven* uses a declarative approach where everything is defined in an *XML* file. *XML* files quickly get big and hard to read. This leads to hard-to-maintain build configurations. In *Gradle*, build files are defined using a domain-specific language (DSL) based either on *Groovy* or *Kotlin*. This domain-specific approach allows for concise build script definitions. Moreover, it is possible to write standard *Groovy* or *Kotlin* code in the build scripts, which makes *Gradle* more powerful than the declarative approach of *Maven*. *Gradle*, combined with *Kotlin-DSL*, provides additional type safety for the build scripts. Since *Kotlin* is also used for implementation, the *Kotlin-DSL* offered by *Gradle* allows us to use the same language for build definitions and the implementation. To summarize, *Gradle* with *Kotlin-DSL* provides a concise, easily maintainable, type-safe way for declaring the build process by simultaneously being more powerful. For these reasons, this work uses *Gradle* together with the *Kotlin-DSL* for dependency and build process management.

7.3 Integrated Development Environment (IDE)

The language selection determines not only the build tool options, but also limits the possible IDE selection. *JetBrains IntelliJ IDEA* is the industry standard when it comes to IDEs in the *JVM* ecosystem. Since it is from the same company which developed *Kotlin* and has excellent *Kotlin* support, it is an easy choice.

7.4 Frameworks

The main framework used in this project is *Spring Boot* [4]. *Spring Boot* is a powerful enterprise-grade framework used for backend web servers and cloud applications. It follows a modular approach and supports many different types of applications. For our use case, the most important features are dependency injection, support for object-relational mapping (ORM) with *Spring Data* and for creating command-line applications with *Spring Shell*.

[2]https://maven.apache.org/
[3]https://gradle.org/
[4]https://spring.io/projects/spring-boot

7.5 Reference Implementation

The reference implementation is used for testing, evaluation, and as a blueprint for platform integrations. It uses an H2[5] database in the backend for data storage. Database access and schema creation is handled by Spring Data JPA[6]. Spring Data JPA is a library provided by the Spring Framework, which reduces the boilerplate code necessary for writing data access layers. It uses Hibernate in the background for mapping classes to database tables and vice versa. Queries can be defined in special repository interfaces either by a naming convention or with the JPA query language. The implementation of these repository interfaces is generated by Spring.

7.6 Fabric Libraries and Tools

This section covers the special libraries and tools necessary for the integration of *Fabric* into the Ledger Access API. As a reminder, such an integration needs two parts, the *Platform Adapter*, and the *LAC*. The adapter is part of the API and responsible for triggering transactions defined by the *LAC*. The *LAC* is installed on the *Fabric* ledger itself. For testing, it is necessary to start a full *Fabric* network with multiple nodes. This is explained in Section 7.6.3.

7.6.1 Fabric Adapter

Fabric provides a library for communication with nodes and contracts installed on them. The library is called *fabric-java-gateway*[7] and provides all the functionality necessary for handling connections, sending/receiving messages, managing wallets, and handling identities.

7.6.2 Fabric Contract

For developing contracts, *Fabric* provides the *fabric-chaincode-shim*[8] library for Java. This library provides all classes, methods, annotations, and helpers necessary for implementing valid *Fabric* contracts.

7.6.3 Fabric Testbed

Fabric is a quite complex platform. Starting multiple nodes and configuring them in a way that they form a useable testnet is not straightforward. The *Fabric* project is aware

[5]https://www.h2database.com/html/main.html
[6]https://spring.io/projects/spring-data-jpa
[7]https://github.com/hyperledger/fabric-gateway-java
[8]https://github.com/hyperledger/fabric-chaincode-java

of this and provides tooling to bring a network up quickly, reliably and repeatably. To achieve this, they use containerization in the form of *Docker*. More information on how to use the provided testnet can be found in the Fabric samples *GitHub* repository.[9]

7.7 Corda Libraries and Tools

Similar to the *Fabric* section before, the *Corda* integration needs an adapter, an *LAC* and a way to start a testnet with multiple nodes. The remainder of this section will describe what libraries and tools are necessary for implementation and testing.

7.7.1 Corda Adapter

The communication between the *Corda* network and a *Corda* client, which is the *LA API* in this case, is carried out over remote procedure calls (RPCs). All the necessary functionality for this communication is provided by the *corda-rpc* and *corda-jackson* libraries.

7.7.2 Corda Contract

Corda contracts are also called *CordApps*. *CordApps* are built using a *Gradle* plugin with the same name provided by the *Corda* project. The main functionality of the plugin is to package all dependencies and modules into a single fat JAR. This JAR is then signed and can be deployed to a node. The plugin also offers the possibility to provide metadata of the CordApp like name, version, licenses, and so on. The libraries needed for *CordApp* implementation are *corda-core*, *corda-node-api*, and *corda*. The documentation and content of these packages can be found on the *Corda* website[10].

7.7.3 Corda Testbed

Corda provides two *Gradle* plugins which allow a test network with multiple nodes, called *Cordform* and *Dockerform*, to be spun up. As the name suggests, the *Dockerform* plugin allows the nodes to be started inside *Docker* containers, similar to *Fabrics* approach. This project uses the *Cordform* plugin, which allows the network participants to be defined directly in the *Gradle* build file. The network is then started by executing the respective Gradle task. No virtualization or containerization is used, the nodes simply run on the host machine and use different directories and ports. More information on how to use these *Gradle* plugins can be found in the official *Corda* documentation[11].

[9] https://github.com/hyperledger/fabric-samples
[10] https://api.corda.net/api/corda-os/
[11] https://docs.corda.net/docs/corda-os/4.6/generating-a-node.html

7.8 Demo Client

The main purpose of the *Demo Client* is to provide an example of how a *Client Application* could use *Contract Applications*. Furthermore, the demo helps during development by providing a command-line interface that allows queries to be executed and transactions to be triggered. This functionality will also be used in the evaluation of the*LA API* to execute end-to-end test scenarios. Its only additional external dependency is to the Spring Shell project. This library provides functionality that makes implementing a command-line interface easy. The user only needs to define a method for each command and the associated parameters. Everything else is handled by the library.

CHAPTER 8

LA API Implementation

In Chapter 5 the architectural design of the *LA API* was introduced. For reference, Figure 5.1 gives an overview of the main components of the proposed API. This chapter focuses on the implementation of platform integrations. The design of the interfaces necessary to integrate a new platform into the *LA API* is described in Section 5.3 in detail. The following is just a short recap. To integrate a new platform, three interface implementations and the *LAC* must be provided. These interfaces are the *LALedgerAccessService*, the *LAParticipantService*, and the *LASignatureService*. The responsibilities of these interfaces are as follows: The *LALedgerAccessService* provides methods for read and write access to the ledger, the *LAParticipant* is responsible for providing access to ledger identities, and the *LASignatureService*'s task is to collect the signatures for transactions. Besides these off-chain modules, it is also necessary to provide an on-chain component called the *LAC*. The *LAC* is a general-purpose contract written utilizing the tools and frameworks provided by the target platform. The main responsibilities of the *LAC* are the reading/writing of the state, transaction validation, consensus enforcement (i.e., checking of signatures), and possibly the gathering of signatures. If the *LAC* can gather the signatures itself, it is not necessary to provide an implementation of the *LASignatureCollector*. Gathering signatures on-chain by the *LAC* is the preferred way; the *LASignatureCollector* should only be used if this is not possible. The remainder of this chapter will be organized as follows. Section 8.1 will introduce a reference implementation which does not use a DLT in the background, but rather a standard database for state storage. After that, Section 8.2 shows the implementation of the Platform Adapter for the Hyperledger Fabric platform. Following that, Section 8.3 describes the Platform Adapter for the Corda platform. The chapter is then concluded with a summary in Section 8.4.

8.1 Reference Adapter

The *Reference Adapter* is an exemplary platform integration. Instead of a real distributed ledger platform, it uses a standard SQL database in the background. Identities are provided by certificates and private keys stored in a Java Keystore. The collection of signatures is simplified, since for this proof-of-concept it is enough to show that signatures can be collected. The way they are collected is an implementation detail and not the focus of this work. The *Reference Adapter* serves multiple purposes. First, it can be used to develop and test new smart contracts using the *LA API* without the need to deploy a test network with multiple nodes. This makes for faster development cycles and less general overhead when building *Contract Applications*. Moreover, as the name implies, it serves as a reference for other *Platform Adapters* to show that they deliver the correct results. Executing the same transactions against the *Reference Adapter* and other adapters must always deliver the same results. The *Reference Adapter* can also be used as an example of a platform integration. A *Platform Integrator* can use it to derive the steps necessary to integrate a platform of choice.

8.1.1 LedgerAccessContract Implementation

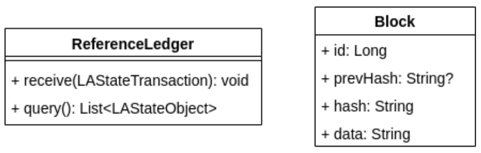

Figure 8.1: Reference LedgerAccessContract Implementation

For the reference implementation, the distributed ledger is replaced by a relational database as data storage and some logic on top to validate the submitted transactions. Figure 8.1 shows the two most important components of the mocked ledger. As the name suggests, the *Block* class on the right represents one block of a blockchain. The properties of this domain object fulfill the following purposes:

- *id*: primary key
- *prevHash*: hash of the previous block; is null for genesis blocks
- *hash*: the hash value of the *data* and *prevHash*
- *data*: JSON representation of the *LAStateObject* contained in the block

This data model is designed to contain multiple blockchains. On a fundamental level, each block without a *prevHash* starts a new chain. Practically starting new chains is limited to transaction types which allow for a *LAStateTransition* with an empty previous state. The *ReferenceLedger* class on the left is responsible for transaction validation and mapping from the *LA API* domain to the *Block* domain described above. It provides two methods, one for querying the ledger state and one for processing an *LAStateTransaction*. When the *receive* function is called, the following steps are executed:

1. Get the appropriate *LAValidator* for the transaction type (see 5.2.4)
2. Validate the *LAStateTransition* contained in the *LAStateTransaction* (see transaction model Section 5.2.2)
3. Verify that all necessary signatures are provided (see consensus model Section 5.2.3)
4. Map the *LAStateTransition* to a new *Block* object
5. Save the *Block* in the database

It should be noted that if any of the validation steps fail, the transaction execution is terminated and no changes to the state are applied. The second method is responsible for querying the database and mapping the result back to the *LA API* domain. Broken down into steps, the logic of the query method can be described as follows:

- Find all genesis blocks, i.e., *Blocks* with no *prevHash*
- Find the whole chain of each genesis block by using the hash of the current block to find the next block until there is no next block
- Deserialize the *data* contained in the last block of each chain
- Return the resulting list of *LAStateObjects*

As described above, the query method returns a list of *LAStateObjects*. Each of these objects represents the current state of one chain and is associated with exactly one *Contract Application* instance.

8.1.2 LAParticipantService Implementation

Similar to the reference implementation of the *LAC*, the complex, distributed permission layer of a real blockchain system is replaced by a simpler, centralized approach. Specifically, the reference implementation uses a Java Keystore for storing identities. For testing purposes, this Keystore is prefilled with several participants. One identity is composed out of an alias, a certificate, and a private key. The method implementations necessary to create an *LAParticipantService* are straightforward and will not be described here in detail. On a high level, the methods use the Java Security API to access the Keystore and load the necessary information. This data is then mapped to the *LA API* domain and returned.

8.1.3 LALedgerAccessService Implementation

An *LALedgerAccessService* implementation must provide two methods: the *submit* and the *query* method. In the case of the *ReferenceAdapter*, these two method implementations are straightforward. They simply call the respective function of the *LAC* and relay the result back to the caller. Since the *LAC* runs locally and the design is fully under our control, no translation/mapping and serialization are required.

8.1.4 LASignatureService Implementation

The *LASignatureService* is responsible for collecting the necessary signatures for a transaction from the participants. For the *ReferenceAdapter*, private keys of each participant are stored locally in the Keystore. If a signature is requested, the private key of the respective participant is loaded and used to sign the transaction. In a real-world scenario, the participant associated with the identity would receive a notification of some form and would then either sign the transaction or not. Depending on the action of the user, the transaction execution can be continued or not. Once all signatures are collected, the transaction is submitted over the *LALedgerAccessService* to the *LAC*.

8.2 Fabric Adapter

Fabric was chosen for integration with the *LA API* for multiple reasons. First of all, it is one of the most widely known permissioned ledger platforms. It is also quite mature and provides many features. Showing that the *LA API* works with it also demonstrates that it works with a mature, widely-used, production-ready platform.

8.2.1 LedgerAccessContract Implementation

The implementation of the Fabric *LAC* uses the Chaincode library (fabric-chaincode-shim) provided by the Fabric project. Figure 8.2 shows the classes of the contract implementation. The diagram can be split vertically and horizontally into four quadrants. Vertically, the diagram is split between classes provided by the Fabric library and the code implemented for the *LA API*. Horizontally, the classes are split between their purpose. The upper classes are responsible for state storage where the classes below the divider contain the business logic of the contract.

For state management, the Fabric-Shim provides the *ChaincodeStub* class. This class provides the core functionality needed for basic CRUD operations on the ledger. Furthermore, it provides several convenience functions which make querying and general interaction with the ledger easier. The *Context* class is a simple utility class which wraps the *ChaincodeStub* and the *ClientIdentity* together. The *ClientIdentity* property is

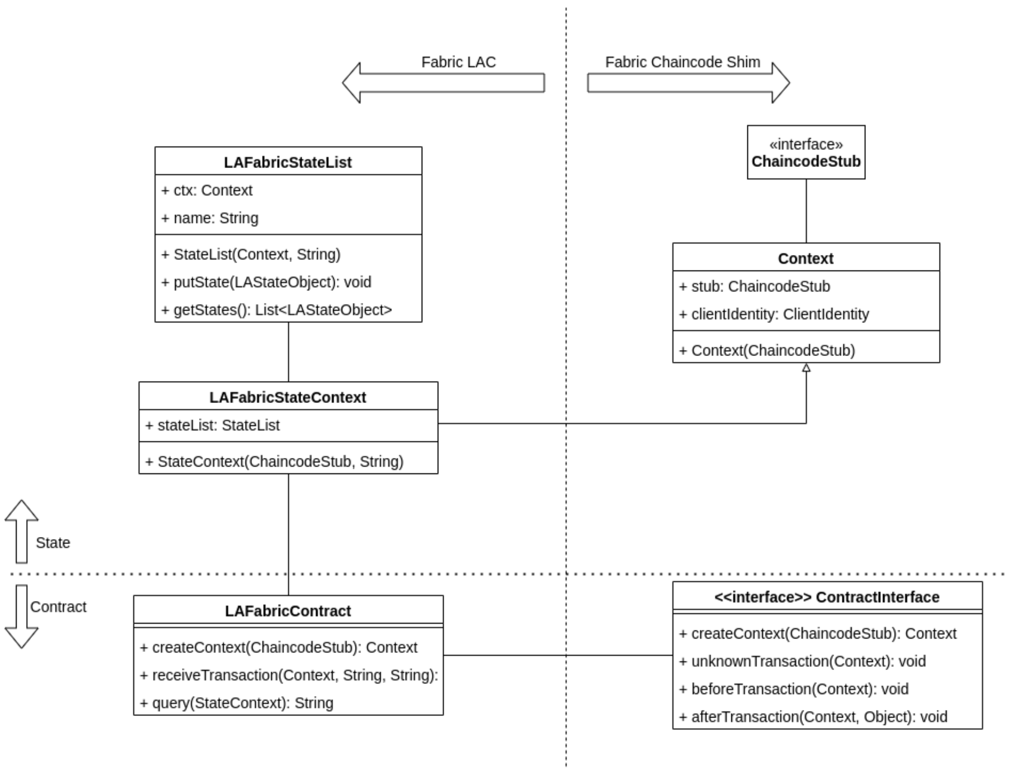
Fabric LAC
Fabric Chaincode Shim
LAFabricStateList
+ ctx: Context
+ name: String
+ StateList(Context, String)
+ putState(LAStateObject): void
+ getStates(): List<LAStateObject>
«interface» ChaincodeStub
Context
+ stub: ChaincodeStub
+ clientIdentity: ClientIdentity
+ Context(ChaincodeStub)
LAFabricStateContext
+ stateList: StateList
+ StateContext(ChaincodeStub, String)
State
Contract
LAFabricContract
+ createContext(ChaincodeStub): Context
+ receiveTransaction(Context, String, String):
+ query(StateContext): String
<<interface>> ContractInterface
+ createContext(ChaincodeStub): Context
+ unknownTransaction(Context): void
+ beforeTransaction(Context): void
+ afterTransaction(Context, Object): void

Figure 8.2: Fabric LedgerAccessContract Implementation

mainly an abstraction over the X509 certificate of the client who submitted the current transaction. Its main purpose is to provide utility functions often needed to make the handling of the certificates easier. All the information provided by the *ClientIdentity* property is acquired from the *ChaincodeStub*, so the *Context* class provides no extra functionality compared to the *ChaincodeStub*; it is just for convenience.

This contract does not use a domain model for storing the state. The *LAStateObject* is simply serialized to its JSON representation. Each state has a key associated with it to make it uniquely identifiable. The *ChaincodeStub* provides support for composite keys. This is convenient for being able to uniquely identify each state, but also makes it possible to retrieve a group of states. For this use case, the composite key is built by combining a constant string called *name* and the *UUID* of the *LAStateObject*, where *name* represents the name of the *Contract Application*. This allows for the retrieval of the state either by its *UUID* or of all states related to a specific *Contract Application* by providing the *name*.

The recommended way to implement custom state handling in a Fabric contract is to subclass the *Context* class. In Figure 8.2 this subclass is the *LAFabricStateContract*. This class acts as a wrapper for the custom *LAFabricStateList* class which handles interaction with the *ChaincodeStub*. It provides the following methods:

1. *putState*: Takes an *LAStateObject*, serializes it to a JSON representation and calls the *putStringState* function of the *ChaincodeStub*. This provided method handles all other steps necessary to write the serialized *LAStateObject* to the state store.
2. *getStates*: The name part of the composite key is used to retrieve all states of this contract. This is done by calling the *getStateByPartialCompositeKey* method of the *ChaincodeStub*. This JSON list of *LAStateObject*s is then deserialized and returned.

The last missing piece for a fully functional Fabric contract is the contract class itself. Each Fabric contract has to implement the *ContractInterface*. The methods declared by this interface can be seen in Figure 8.2. All these methods provide a default implementation. This means that they are optional and do not need to be overridden in the implementing class. The implementation of the *ContractInterface* is called *LAFabricContract*. The only method needed for this implementation is the *createContext* method. This method is called by Fabric itself and provides the contract access to the *ChaincodeStub*. It returns an instance of *LAFabricStateContext*, which is then automatically provided as the first parameter of every transaction method. Transaction methods are the methods of the contract which can be called externally from a client. The *receiveTransaction* and *query* methods are both transaction methods. To turn a normal method into a transaction method, the Fabric-Shim library provides a *Transaction* annotation which is simply put on the method. The two transaction methods implement the following logic:

- *query*: Receives the *LAFabricStateContext* object, accesses its *LAFabricStateList* object and calls *getStates* on it to retrieve the stored *LAStateObject*s. This result is then returned.
- *receiveTransaction*: Receives the *LAFabricStateContext*, and the JSON string representation of the *LATransaction* to be processed. The first step is to deserialize the transaction and retrieve the associated *LAValidator*. Then the *validation* function is executed, and the signatures are validated. If all checks pass, the *LAFabricStateContext* parameter is used to access the *LAFabricStateList* and call the *putState* method on it. The state to be persisted is the new one contained in the *LAStateTransaction*.

8.2.2 LAParticipantService Implementation

The Fabric Gateway library, which is used by clients to communicate with a Fabric network, provides the concept of *Wallets*. *Wallets* in *Fabric* are similar to the Keystore used in the reference implementation. It serves as a store for certificates and private keys. Fabric's *Wallet* abstraction allows for different storage backends. For this project, we use a *FilesystemWallet* which persists the data to a file on the local filesystem. For testing purposes and simplicity, multiple identities are created automatically at the startup of the test network. All the identities are stored in the same wallet. This makes the implementation of the *LASignatureService* for *Fabric* straightforward, because all private keys necessary can be loaded from the same wallet. The implementation of the *LAParticipantService* for Fabric is also straightforward. The keys are simply loaded from the wallet and mapped to *LAPrivateKey* or *LAParticipant*, depending on which method was called.

8.2.3 LALedgerAccessService Implementation

Compared to the *LALedgerAccessService* of the *ReferenceAdapter*, the one for *Fabric* involves some additional steps. Instead of being able to call the contract directly, the *Fabric Gateway* library is used to communicate with the *LAC* installed on a node on the *Fabric* network. This includes an additional serialization/deserialization step when sending/receiving data to/from the *LAC*. Apart from that, everything works similarly to the *ReferenceAdapter*. The *submit* method collects the necessary signatures utilizing the *LASignatureService*, builds the *LATransaction* object, serializes it, and triggers the respective transaction on the ledger. The *query* function simply triggers the query transaction provided by the *LAC*, deserializes the result, and returns it.

8.2.4 LASignatureService Implementation

The signature collection and therefore the implementation of the *LASignatureService* for *Fabric* works similarly to the one from the *ReferenceAdapter*. When a signature is requested, the *LAParticipantService* is used to retrieve the identity and the transaction is signed. This means that all transactions are signed automatically and the participant is never asked for confirmation. Again, this is only for simplifying the proof-of-concept. In a real-world scenario, the participant would have to agree explicitly to the transaction.

8.3 Corda Adapter

The *Corda* platform was, similarly to *Fabric*, chosen because it is a mature and widely known platform. In addition, it uses vastly different concepts and techniques to deliver the fundamental functionality of a distributed ledger. The most important difference is that it uses a UTXO model and does not distribute transaction data to all nodes in the network by default. Showing that the *LA API* can provide a single interface to such a wide variety of platforms underlines its usefulness and flexibility.

8.3.1 LedgerAccessContract Implementation

The proposed *LAC* for *Corda* has three main components: a state, a contract, and a flow. Each state declared with the *Corda* framework for Java requires the *BelongsToContract*

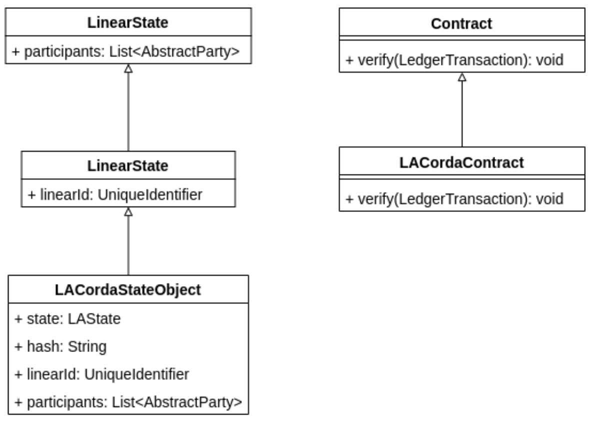
LinearState
+ participants: List<AbstractParty>
Contract
+ verify(LedgerTransaction): void
LinearState
+ linearId: UniqueIdentifier
LACordaContract
+ verify(LedgerTransaction): void
LACordaStateObject
+ state: LAState
+ hash: String
+ linearId: UniqueIdentifier
+ participants: List<AbstractParty>

Figure 8.3: Corda LedgerAccessContract Implementation

annotation and must implement one of the provided state interfaces. For the *LAC* we use the *LinearState* interface, which represents a state that evolves by superseding itself. Meaning that each state has exactly one predecessor and exactly one successor. These special requirements are the reason why the *LAStateObject* class cannot be directly reused and a new class for representing the *LAStateObject* has to be created. The name of the class is *LACordaStateObject* and its structure can be seen in Figure 8.3. It is the same as the *LAStateObject*, except for the added properties required by the *LinearState* interface. Both the *linearId* and the *participants* are mapped from the referenced *LAStateObject*. The *UUID* is used for the *linearId* and the *participants* are the same as the *participants* of the *LAState*.

A contract in *Corda* is a class implementing the *Contract* interface provided by *Corda* and shown in Section 8.3. The interface declares a single function called *verify* and takes a *LedgerTransaction* as a parameter. The purpose of this function is to enforce the state transition rules. This is conceptually the same as the *validateTransaction* function of *LAValidator*. For the *LAC*, the *verify* function executes the following steps:

1. Map the data of the *LedgerTransaction* input parameter to an *LATransition*
2. Retrieve the *LAValidator* for the given transaction type
3. Execute the *validateTransaction* function of the validator

In *Corda*, contracts are executed by a client initiating a *Flow*. *Flows* are written using the *Corda Flow Framework* and declare the steps a transaction has to go through to be considered valid and recorded on the ledger. For a more detailed explanation of *Corda Flows* and *Corda* in general, see Section 2.9. In the case of the *LAC*, the *Flow* runs through the following steps:

1. The flow is initiated and receives an *LAStateTransition* as a parameter.

2. The *LAStateTransition* is transformed into a *CordaTransaction* with the following steps:

   a) If the *LAStateTransition* has an *oldState*, use the *UUID* to retrieve the *oldState* from the ledger and set it as the input state of the transaction.

   b) Map the *newState* to an *LACordaStateObject* and set it as the output of the transaction

   c) Map the *LACommand* to the provided *Command* interface and add it to the transaction

3. The transaction is signed by the current node.

4. Then a flow session is created for each participant listed in the command.

5. These flow sessions are then used to request the signatures of the participants.

6. Once all flow sessions return the signatures, the transaction is finalized and recorded on the ledger.

The steps above only describe those executed by the initiator. Since the initiator triggers the flow on the nodes of other participants, there has to be a second flow that replies to these invocations. For this proof-of-concept, this flow simply signs all transactions containing a single output of the type *LACordaStateObject*. Real-world applications can declare a special signing logic here. The last step of this flow is to create a subflow for waiting on the finalization of the transaction. As soon as it is received, the state is recorded on the ledger.

Note that the *LAC* for *Corda* does not implement any logic for externally accessing the ledger state. This is because Corda provides mechanisms to query the ledger over its client library without the need to implement it in the contract.

8.3.2 LAUserService Implementation

The implementation of the *LAUserService* for *Corda* is also straightforward. *Corda* provides the *Party* type to refer to participants. A *Party* object contains a name and a public key and can therefore easily be mapped to an *LAParticipant*. Retrieving the current logged-in identity is done by an RPC call. All other identities can be retrieved from the network map, which is also accessed via an RPC call.

8.3.3 LALedgerAccessService Implementation

From an high-level viewpoint, the *LALedgerAccessService* for *Corda* does basically the same things as the other implementations. It provides implementations for the *query* and *submit* methods, which communicate with a *Corda* network. The communication is handled by the client framework *Corda* provides. When a new transaction is submitted to the *LALedgerAccessService*, it translates the method input to the *Corda* domain and starts the *Initiator Flow*. This *Flow* was introduced and described in Section 8.3.1. The *query* implementation of the *LALedgerAccessService* interface utilizes the *vaultQuery* RPC call provided by Corda to retrieve all objects of type *LACordaStateObject* from the ledger. This data is then mapped to an instance of *LAStateObjects* and returned to the caller.

8.3.4 LASignatureService Implementation

For *Corda* it is not necessary to provide an implementation for the *LASignatureService*, because collecting signatures is supported by the *Corda Flow Framework*. The way the signatures are collected was described in Section 8.3.1.

8.4 Summary

This chapter showed how different platforms can be integrated into the *LA API*, using *Fabric* and *Corda* as examples. It explained the implementations of the contracts which run on the chain (*LAC*) and their counterpart on the application side (*LPI*). Moreover, a reference implementation was created to define the expected behavior of the real platform integrations. The reference implementation can also be used for rapid prototyping, because it allows for smart contract development without the need for a test network. Since a correct platform integration must always return the same result as the reference implementation, there should be no surprises when deploying the contract tested with the reference implementation on a real network.

CHAPTER 9

Use Case Implementation

This chapter explains the implementation of the use case introduced in Chapter 6 utilizing the *LA API*. The implementation follows the model introduced in Section 5.2. The general idea behind the use case, the purpose of the classes and their properties, the transaction types, and their rules were already explained in detail there. The implementation is mostly straightforward, but several interesting details will be discussed in the upcoming sections.

9.1 State Model

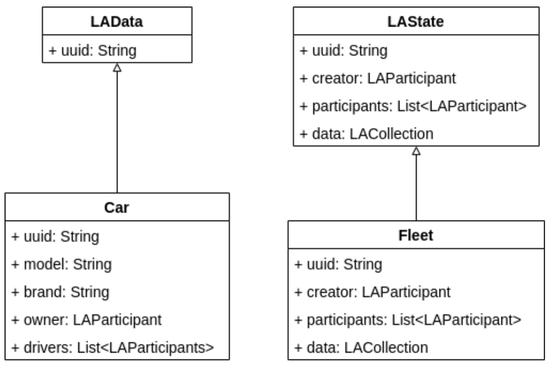

Figure 9.1: State Model of the Fleet Contract

As described in Section 5.2.1, a contract implemented using the *SCI* needs to provide two classes for the state model: an implementation of *LAData* and an implementation of *LAState*. To recap, *LAData* represents a single object and declares the type of object or asset that should be managed by the contract. The *LAState* class, on the other hand, represents a collection of *LAData* elements together with the information of the permission layer like the creator, the owner, and the other participants. Figure 9.2 shows the base classes of the state model together with their subclasses used for the fleet application. It can easily be seen that the fleet state model is a straightforward application of the requirements defined in Section 6.2 to the model introduced in Section 5.2.1.

9.2 Transaction Model

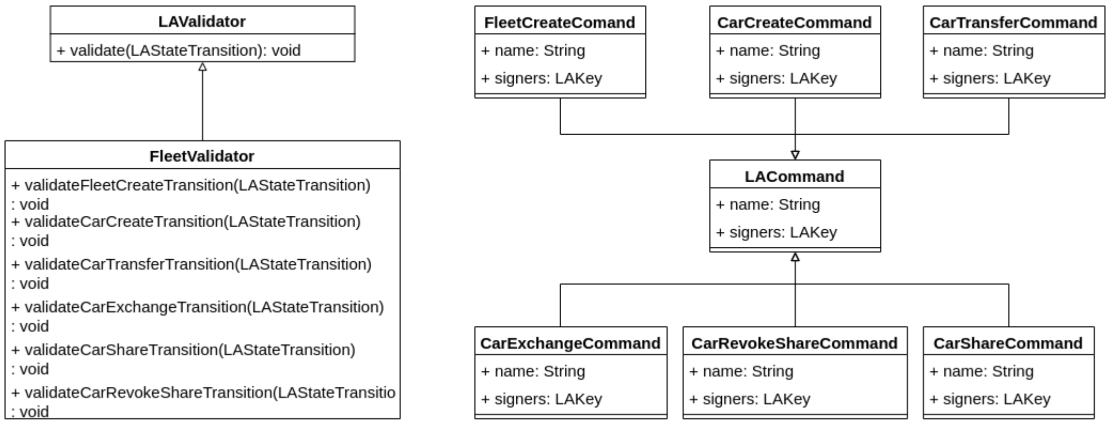
LAValidator
+ validate(LAStateTransition): void
FleetValidator
+ validateFleetCreateTransition(LAStateTransition) : void
+ validateCarCreateTransition(LAStateTransition) : void
+ validateCarTransferTransition(LAStateTransition) : void
+ validateCarExchangeTransition(LAStateTransition) : void
+ validateCarShareTransition(LAStateTransition) : void
+ validateCarRevokeShareTransition(LAStateTransitio : void
FleetCreateComand
+ name: String
+ signers: LAKey
CarCreateCommand
+ name: String
+ signers: LAKey
CarTransferCommand
+ name: String
+ signers: LAKey
LACommand
+ name: String
+ signers: LAKey
CarExchangeCommand
+ name: String
+ signers: LAKey
CarRevokeShareCommand
+ name: String
+ signers: LAKey
CarShareCommand
+ name: String
+ signers: LAKey

Figure 9.2: Transaction Model of the Fleet Contract

An implementation of the transaction model, explained in detail in Section 5.2.2, needs to define a validator and the transaction types in the form of commands. The validator must support validating all declared transaction types. The rules of the transition which have to be enforced by the validator are explained in Section 6.3. Like the state model, the transaction model for the fleet application is a straightforward application of the *SCI* transaction model to the requirements declared by the use case definition.

9.3 Consensus Model

The consensus model describes which transactions have to be signed by whom. The exact rules for the fleet use case were given in Section 6.4. Enforcement of these rules is performed by the transaction validation step, which is part of the *LAC*. During the transaction generation, the required signers of the transaction are set. Which signers are required is determined by the consensus rules. For the fleet use case, a helper class, which generates transactions for each transaction type, given the required input parameters, was created. The factory sets the required signers for each transaction according to the consensus rules of the use case.

CHAPTER 10

Demo Client Application

As a reminder, Figure 5.1 shows an overview of the *LA API* and its surrounding applications. These are *Platform Adapters*, *Contract Applications*, and *Client Applications*. Example implementations for *Platform Adapters* were given in Chapter 8, whereas an example for a *Contract Application* was given in Chapter 9. The last missing part for a fully functional example of an application using the *LA API* is a *Client Application*. To complete the implementation part of this thesis, this chapter will introduce the *Demo Client*. A simple command-line application, the *Demo Client* uses the *Contract Application* created in Chapter 9 to manage a fleet of cars with any of the *Platform Adapters* created in Chapter 8. Besides completing the example given in this thesis and serving as a template for developers who want to implement their use case with the *LA API*, the *Demo Client* will also aid in the functional evaluation of the proposed API itself. For that, the *Demo Client* will be used in Chapter 11 to trigger the same transactions on different platforms. The results of the executions on different platforms will then be compared to show that the results are the same. This will prove that the design and implementation of the *LA API* works as intended.

10.1 Architecture

The Demo Client utilizes the same technology stack as the *LA API* itself. It is primarily implemented with Spring Boot in Kotlin. Users interact with the application over a command-line interface, which is created with the help of the Spring Shell[1] library. Annotation-based Spring configuration is used to provide the platform-based services. This means that a simple configuration change allows the platform used in the backend to be changed. All platform adapters introduced in Chapter 8 (Reference, Fabric, Corda)

[1] https://spring.io/projects/spring-shell

are supported. In addition, the fleet contract application designed in Chapter 6 and implemented in Chapter 9 is used to demonstrate the full functionality of the *LA API*.

10.2 Available Commands

A user can trigger transactions provided by the *Fleet* contract application by submitting a command over the command-line interface. The implementation of these commands is quite simple. The parameters of the command are handed over to the respective function of the transaction factory provided by the *Contract Application*. The result is a fully populated *LATransition* object which is submitted to the ledger via the *submit* function provided by the *LALedgerAccessService*. The only exception is the query command, which calls the *query* method of the service directly and outputs the results. The following is a list of all available commands and their parameters.

- **query** - Queries all available fleets and logs them in a nicely formatted way to the console
- **create-fleet** - Creates a new fleet
- **create-car** - Adds a new car to an existing fleet
  - **fleetUUID:** The UUID of the targeted fleet
  - **brand:** The brand of the car to create (e.g., Audi)
  - **model:** The model of the car to create (e.g., A6)
- **transfer-car** - Transfers the ownership of a car to another user
  - **fleetUUID:** The UUID of the targeted fleet
  - **carUUID:** The UUID of the car to transfer
  - **newOwnerAlias:** The alias of the new owner
- **exchange-car** - The owners of two cars are exchanged.
  - **fleetUUID:** The UUID of the targeted fleet
  - **car1UUID:** The UUID of the first car
  - **car2UUID:** The UUID of the second car
- **share-car** - Adds a new driver to the car
  - **fleetUUID:** The UUID of the targeted fleet
  - **carUUID:** The UUID of the car to transfer
  - **newDriverAlias:** The alias of the new driver

- **revoke-share-car** - Removes a driver from the car
  - **fleetUUID:** The UUID of the targeted fleet
  - **carUUID:** The UUID of the car to transfer
  - **driverToRemoveAlias:** The alias of the driver to remove

CHAPTER 11

Evaluation

The evaluation of the *LA API*'s design and implementation is aimed to prove that the following two statements are true.

- All requirements of all stakeholders defined in Chapter 4 are fulfilled.
- Executing a series of transactions provided by a *Contract Application* via a *Client Application* results in the same state, regardless of which platform is used in the backend.

Showing that these two statements are true will demonstrate that the *LA API* fulfills all requirements in theory and that a working prototype, which shows the validity of the concept in practice, exists. Proving these two statements will be approached in the following way: The first statement will be proven by going through all the requirements and reasons for their fulfillment. This will be done in Section 11.1. The second statement will be proven by a functional evaluation in Section 11.2. A sequence of transactions covering all requirements of the *Contract Application Programmer* role will be defined. This sequence will be executed against all three platform integrations introduced in Chapter 8. This execution, if correct, covers all functional requirements of the *Platform Integrator* and *Application Programmer* roles. Showing that the execution on *Corda* and *Fabric* delivers the same result as that of the reference implementation will prove that the *LA API* concept works not only in theory, but is technically feasible in practice. The chapter will be concluded in Section 11.3, where the results of the evaluation are discussed and summarized.

11.1 API Design Evaluation

In Chapter 4, the requirements on the API design were formulated as user stories from the viewpoint of three roles: *Platform Integrator (PI)*, *Contract Application Programmer (CP)*, and *Application Programmer (AP)*. This section will go through every requirement of every role and will explain how and where in the work it was fulfilled, starting with the requirements of the **Platform Integrator:**

- **PI1**: Chapter 8 shows how new platforms are integrated by implementing the interfaces of the *LA API* for *Fabric* and *Corda*.
- **PI2**: Unified access to the permission layer of integrated platforms is provided by the implementations of the *LAParticipantService*. Three examples for such implementations were given in Chapter 8.
- **PI3**: The implementations given in Chapter 8 show that integrating new platforms can be done easily with the standard tooling described in Chapter 7.

Next, the requirements of the **Contract Application Programmer** are checked:

- **CP1**: The *Car* data object of the state model in Section 9.1 represents a custom asset type and shows that the *LA API* allows for the creation of custom asset types with arbitrary fields.
- **CP2**: The *Fleet* class of the use case implementation in Section 9.1 represents a collection of custom asset types. This collection can be stored on the ledger by executing a minting transaction.
- **CP3**: Adding new assets to the ledger is done with the "Create Car" transaction provided by the use case implementation.
- **CP4**: An example for transferring assets is given by the *Transfer Car* transaction of the use case.
- **CP5**: The *Exchange Car* transaction defined and implemented by the given use case exemplifies an exchange operation.
- **CP6**: The *Demo Client* introduced in Chapter 10 shows that its possible to execute the transaction of the same *Contract Application* on different platforms. Therefore, the *Reference Adapter* can be used during development and an adapter for a real platform can be used in production.

The requirements of the **Application Programmer** were fulfilled as follows:

- **AP1**: The *Demo Client* presented in Chapter 10 shows how to execute transactions provided by a *Contract Application*.
- **AP2**: The *Demo Client* is able to use the Reference, Fabric, or Corda Adapters to execute transactions on the respective platform.
- **AP3**: The same reasoning as for *SP6*. The *Demo Client* shows that the reference adapter can be used in place of any other adapter during development.

The explanations given above show that, in theory, all the requirements are fulfilled. What is missing is to show that everything is working when executing the solution in practice. This is the topic of the next section.

11.2 Implementation Evaluation

This section will show that the design and implementation of the *LA API* is functionally complete by executing a sequence of transactions on all available platforms. The transactions used for this evaluation are the ones introduced for the *Fleet Management* use case given in Chapter 6. The transactions will be triggered by utilizing the command-line interface of the *Demo Client* presented in Chapter 10. After each transaction, the query command will be executed to retrieve the current state of the contract application. Each of these intermediate results will be logged and has to match the results of executions of the same transactions on other platforms.

11.2.1 Transaction Sequence

The transaction sequence chosen for this functional evaluation covers all transaction types defined by the *Fleet Management* use case. The following list describes the commands sent to the *Demo Client*, which trigger the respective transaction:

1. Create Fleet
2. Create Car - BMW, 530d // Car1
3. Create Car - Audi, A6 // Car2
4. Transfer Car - <FleetUUID> <Car1UUID> party2
5. Transfer Car - <FleetUUID> <Car2UUID> party3
6. Exchange Car - <FleetUUID> <Car1UUID> <Car2UUID>
7. Share Car - <FleetUUID> <Car1UUID> party4

8. Revoke Car Sharing - <FleetUUID> <Car1UUID> party4

In short, this transaction sequence can be worded in the following way. A fleet with two cars is created. One is a BMW 530d and the other is an Audi A6. Both of these cars are then transferred to new owners. The next step swaps the owner of the two cars. Then, the BMW is shared with party4. The last transaction revokes the driving right of party4.

11.2.2 Reference Results

Executing the transaction sequence from the previous section on the *Reference Implementation* gave the following results. For clarity, hash values are shortened to show only the last five digits. Moreover, participant objects like creator and owner are represented by their aliases. In reality, these objects also have a public key object associated with them.

shell:> create-fleet

```
{
    "state": {
        "type": "Fleet",
        "uuid": "7153f261-268d-4bdf-a462-566d45884971",
        "creator": "party1",
        "data": [
        ]
    },
    "hash": "...779de"
}
```

shell:> create-car 7153f261-268d-4bdf-a462-566d45884971 BMW 530d

```
{
    "state": {
        "type": "Fleet",
        "uuid": "7153f261-268d-4bdf-a462-566d45884971",
        "creator": "party1",
        "data": [
            {
                "data": {
                    "type": "Car",
                    "uuid": "5baaf767-9f50-4372-a8e2-6e800d5575
                        ed",
                    "model": "530d",
                    "brand": "BMW",
                    "owner": "party1",
                    "drivers": []
                },
                "hash": "...81eca"
            }
        ]
    },
    "hash": "...684b4"
}
```

shell:> create-car 7153f261-268d-4bdf-a462-566d45884971 Audi A6

```
[
    {
        "state": {
            "type": "Fleet",
            "uuid": "7153f261-268d-4bdf-a462-566d45884971",
            "creator": "party1",
            "data": [
                {
                    "data": {
                        "type": "Car",
                        "uuid": "5baaf767-9f50-4372-a8e2-6e800d
                           5575ed",
                        "model": "530d",
                        "brand": "BMW",
                        "owner": "party1",
                        "drivers": []
                    },
                    "hash": "...81eca"
                },
                {
                    "data": {
                        "type": "Car",
                        "uuid": "cee5fa7b-856a-4e30-b7c9-dcbe90
                           56abf6",
                        "model": "A6",
                        "brand": "Audi",
                        "owner": "party1",
                        "drivers": []
                    },
                    "hash": "...3726c"
                }
            ]
        },
            "hash": "...2b2b6"
        }
]
```

shell:> transfer-car 7153f261-268d-4bdf-a462-566d45884971 5baaf767-9f50-4372-a8e2-6e800d5575ed party2

```
{
    "state": {
        "type": "Fleet",
        "uuid": "7153f261-268d-4bdf-a462-566d45884971",
        "creator": "party1",
        "data": [
            {
                "data": {
                    "type": "Car",
                    "uuid": "5baaf767-9f50-4372-a8e2-6e800d5575
                        ed",
                    "model": "530d",
                    "brand": "BMW",
                    "owner": "party2",
                    "drivers": []
                },
                "hash": "...72b46"
            },
            {
                "data": {
                    "type": "Car",
                    "uuid": "cee5fa7b-856a-4e30-b7c9-dcbe9056
                        abf6",
                    "model": "A6",
                    "brand": "Audi",
                    "owner": "party1",
                    "drivers": []
                },
                "hash": "...3726c"
            }
        ]
    },
    "hash": "...a565e"
}
```

shell:> transfer-car 7153f261-268d-4bdf-a462-566d45884971 cee5fa7b-856a-4e30-b7c9-dcbe9056abf6 party3

```
{
    "state": {
        "type": "Fleet",
        "uuid": "7153f261-268d-4bdf-a462-566d45884971",
        "creator": "party1",
        "data": [
            {
                "data": {
                    "type": "Car",
                    "uuid": "5baaf767-9f50-4372-a8e2-6e800d5575
                       ed",
                    "model": "530d",
                    "brand": "BMW",
                    "owner": "party2",
                    "drivers": []
                },
                "hash": "...2b46"
            },
            {
                "data": {
                    "type": "Car",
                    "uuid": "cee5fa7b-856a-4e30-b7c9-dcbe9056
                       abf6",
                    "model": "A6",
                    "brand": "Audi",
                    "owner": "party3",
                    "drivers": []
                },
                "hash": "...35e41"
            }
        ]
    },
    "hash": "...26b2d"
}
```

shell:> exchange-car 7153f261-268d-4bdf-a462-566d45884971 5baaf767-9f50-4372-a8e2-6e800d5575ed cee5fa7b-856a-4e30-b7c9-dcbe9056abf6

```
{
    "state": {
        "type": "Fleet",
        "uuid": "7153f261-268d-4bdf-a462-566d45884971",
        "creator": "party1",
        "data": [
            {
                "data": {
                    "type": "Car",
                    "uuid": "5baaf767-9f50-4372-a8e2-6e800d5575
                       ed",
                    "model": "530d",
                    "brand": "BMW",
                    "owner": "party3",
                    "drivers": []
                },
                "hash": "...8e404"
            },
            {
                "data": {
                    "type": "Car",
                    "uuid": "cee5fa7b-856a-4e30-b7c9-dcbe9056
                       abf6",
                    "model": "A6",
                    "brand": "Audi",
                    "owner": "party2",
                    "drivers": []
                },
                "hash": "...f77fa"
            }
        ]
    },
    "hash": "...e78c4"
}
```

shell:> share-car 7153f261-268d-4bdf-a462-566d45884971 5baaf767-9f50-4372-a8e2-6e800d5575ed party4

```
{
    "state": {
        "type": "Fleet",
        "uuid": "7153f261-268d-4bdf-a462-566d45884971",
        "creator": "party1",
        "data": [
            {
                "data": {
                    "type": "Car",
                    "uuid": "5baaf767-9f50-4372-a8e2-6e800d5575
                       ed",
                    "model": "530d",
                    "brand": "BMW",
                    "owner": "party3",
                    "drivers": ["party4"]
                },
                "hash": "...0e815"
            },
            {
                "data": {
                    "type": "Car",
                    "uuid": "cee5fa7b-856a-4e30-b7c9-dcbe9056
                       abf6",
                    "model": "A6",
                    "brand": "Audi",
                    "owner": "party2",
                    "drivers": []
                },
                "hash": "...f77fa"
            }
        ]
    },
    "hash": "...5fdb9"
}
```

shell:> revoke-share-car 7153f261-268d-4bdf-a462-566d45884971 5baaf767-9f50-4372-a8e2-6e800d5575ed party4

```
{
    "state": {
        "type": "Fleet",
        "uuid": "7153f261-268d-4bdf-a462-566d45884971",
        "creator": "party1",
        "data": [
            {
                "data": {
                    "type": "Car",
                    "uuid": "5baaf767-9f50-4372-a8e2-6e800d5575
                       ed",
                    "model": "530d",
                    "brand": "BMW",
                    "owner": "party3",
                    "drivers": []
                },
                "hash": "...0e815"
            },
            {
                "data": {
                    "type": "Car",
                    "uuid": "cee5fa7b-856a-4e30-b7c9-dcbe9056
                       abf6",
                    "model": "A6",
                    "brand": "Audi",
                    "owner": "party2",
                    "drivers": []
                },
                "hash": "...f77fa"
            }
        ]
    },
    "hash": "...5fdb9"
}
```

11.2.3 Fabric and Corda Results

To execute the transaction on a *Corda* and *Fabric* network, virtual test networks were started. These test networks were created utilizing the tooling described in Chapter 7. The evaluation of the executions confirms that all *Platform Adapter*s returned the same results. Since UUIDs and hashes are generated for each execution, these values are different, but this is expected and desired behavior. The data fields themselves match between executions. This shows not only that the *Platform Adapters* were implemented correctly, but also that the design of the *LA API* is functionally complete.

11.3 Conclusion

Section 11.1 showed that the LA API fulfills all requirements set at the beginning of this work (see Chapter 4). In Section 11.2, a functional evaluation was performed to show that the defined requirements are not only supported in theory, but that the implemented use case works as expected. This was done by defining a transaction sequence that executes every supported transaction type. It was shown that the execution of this sequence on *Corda* and *Fabric* delivered the same results as the execution on the *Reference Implementation*. It can therefore be concluded that the design of the *LA API* delivers the expected functionality in theory and practice.

CHAPTER 12

Future Work

Although it was shown that the *LA API* is fully functional and fulfills all requirements, there are still open questions that go beyond the scope of this thesis. The following is a comprehensive list of these questions.

Security Evaluation: It is not clear whether the security properties of the underlying blockchain platform hold. There needs to be a full analysis to guarantee that things like double spending are prevented. The main question here is if the dynamic loading of the transaction validation logic makes it vulnerable to unexpected attacks.

Consensus Model and Transaction Generation: Another potential problem of the *LA API* stems from the fact that the enforcement of the consensus model is dependent on the code which generates the transaction. The consensus model describes who has to sign a transaction for it to be valid. Since the participants who have to sign are selected during the transaction generation, it is essential that transactions are generated by a trusted source. This thesis does not focus on how to make sure that the code comes from a trusted source. For this reason, further work has to be done to ensure that transactions come from a trusted source and cannot be manipulated.

Performance Evaluation: The question here is whether the additional abstraction layer of the *LA API* brings a notable performance hit with it. It is not expected that a huge performance hit will be detected, since only the client API provided by the client is abstracted, but a thorough evaluation should be done to prove the assumption.

Redundancy: The *LA API* could be expanded to support the use of multiple platforms at the same time. This could provide higher consistency, security, and safety against network failures.

Signature Collection: The prototype shown in this thesis used a mock service that signed all transactions in the name of any test users. In practice, the participants have to be contacted and sign the transaction if they agree. Because there are multiple ways to implement such a messaging system, the possibilities have to be analyzed and the most fitting solution has to be determined.

Production Ready: Since only a prototype has been implemented in this thesis, future work should also concentrate on getting the prototype production ready. This includes writing detailed usage documentation, implementing a way for distributed signature collection, and releasing the code as an open-source project with an appropriate license.

Unified Deployment and Maintenance: This work focused on platform-independent smart contracts. The deployment and maintenance of a distributed ledger network still need platform-specific knowledge. Future work could investigate whether it is possible to abstract these activities so that one API can deploy networks running different blockchain platforms. An example of an attempt to achieve this can be found in [LXL+19].

CHAPTER 13

Conclusion

The work set out to achieve mainly one goal: To show that platform-agnostic smart contracts are possible without complicated, hard-to-use, non-standard tooling or without having to write complex code generators.

This goal was achieved by designing an API called the *LA API*. The *LA API* can be split into two parts. One is an abstraction over the client APIs provided by the platforms, called the *LPI*. The *LPI* enables unified access to the permission layer of the platforms. Besides that, the *LPI* provides an interface to communicate with a platform-specific multi-purpose contract (*LAC*) installed on a node of the target network. This contract receives the transactions submitted over the *LPI*, validates them, and, if successful, records the new state on the ledger. The second part is the *Smart Contract Interface*, which is mainly concerned with transaction definition and validation. In other words, it allows for the definition of the data structure and business logic of a smart contract. This is done by providing an implementation for the state, transaction, and consensus model defined by the *SCI*. The *LPI* and *Contract Application* implemented with the *SCI* are then used by *Client Applications* to trigger transactions on a blockchain in a transparent way. At the time of development, the *Client Application* does not need to know which platform will be used.

Showing that the proposed solution works required multiple steps. First, a reference platform integration that uses a relational database instead of a blockchain was implemented. From there, platform integration for the popular blockchain platforms *Corda* and *Fabric* was created. The next step was to design a use case which, when implemented, shows that the *LA API* is powerful enough to implement a wide range of asset management applications. This use case was then implemented using the *LA API*. The last missing piece to perform a full evaluation was the *Demo Client*, an exemplary *Client Application* which supports the sending of the implemented use case transactions to a configured blockchain network.

With all parts implemented, a two-step evaluation was performed. The first step involved a user story-based evaluation which showed that all requirements defined at the beginning of the work were fulfilled. The functional evaluation was performed as an end-to-end test of components on multiple platforms and the results were compared. It was found that all executions all on all platforms (Reference, Corda, and Fabric) delivered the same results. It can be concluded that the *LA API* is an abstraction powerful enough to implement the desired use cases. Moreover, the API was tested with a prototype, so there is no doubt about the feasibility of a production-ready *LA API* implementation.

List of Figures

List of Tables

Bibliography

[B+02] Adam Back et al. Hashcash - a denial of service counter-measure. 2002.

[B+14] Vitalik Buterin et al. A next-generation smart contract and decentralized application platform. *White paper*, 3(37), 2014.

[BG19] Vitalik Buterin and Virgil Griffith. Casper the friendly finality gadget, 2019.

[BSS+20] Akashdeep Bhardwaj, Syed Bilal Hussian Shah, Achyut Shankar, Mamoun Alazab, Manoj Kumar, and Thippa Reddy Gadekallu. Penetration testing framework for smart contract blockchain. *Peer-to-Peer Networking and Applications*, pages 1–16, 2020.

[BZW+19] Qianlan Bai, Xinyan Zhou, Xing Wang, Yuedong Xu, Xin Wang, and Qingsheng Kong. A deep dive into blockchain selfish mining. In *ICC 2019 - 2019 IEEE International Conference on Communications (ICC)*, pages 1–6, 2019.

[CAVG18] Miguel Pincheira Caro, Muhammad Salek Ali, Massimo Vecchio, and Raffaele Giaffreda. Blockchain-based traceability in agri-food supply chain management: A practical implementation. In *2018 IoT Vertical and Topical Summit on Agriculture - Tuscany (IOT Tuscany)*, pages 1–4, 2018.

[CRS+18] Olivia Choudhury, Nolan Rudolph, Issa Sylla, Noor Fairoza, and Amar Das. Auto-generation of smart contracts from domain-specific ontologies and semantic rules. In *2018 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 963–970, 2018.

[dam] Daml programming language.

[DDDN19] Vimal Dwivedi, Vipin Deval, Abhishek Dixit, and Alex Norta. Formal-verification of smart-contract languages: A survey. In Mayank Singh, P.K. Gupta, Vipin Tyagi, Jan Flusser, Tuncer Ören, and Rekha Kashyap, editors, *Advances in Computing and Data Sciences*, pages 738–747, Singapore, 2019. Springer Singapore.

[DMH17] Vikram Dhillon, David Metcalf, and Max Hooper. *The DAO Hacked*, pages 67–78. Apress, Berkeley, CA, 2017.

[Eid20] Christian Eid. *Model-driven approach to smart contract development with automatic code generation.* PhD thesis, Notre Dame University-Louaize, 2020.

[EP19] Joshua Ellul and Gordon Pace. Towards a unified programming model for blockchain smart contract dapp systems. In *2019 38th International Symposium on Reliable Distributed Systems Workshops (SRDSW)*, pages 55–56, 2019.

[FBD$^+$20] Ghareeb Falazi, Uwe Breitenbücher, Florian Daniel, Andrea Lamparelli, Frank Leymann, and Vladimir Yussupov. Smart contract invocation protocol (scip): A protocol for the uniform integration of heterogeneous blockchain smart contracts. In Schahram Dustdar, Eric Yu, Camille Salinesi, Dominique Rieu, and Vik Pant, editors, *Advanced Information Systems Engineering*, pages 134–149, Cham, 2020. Springer International Publishing.

[FHB$^+$19] Ghareeb Falazi, Michael Hahn, Uwe Breitenbücher, Frank Leymann, and Vladimir Yussupov. Process-based composition of permissioned and permissionless blockchain smart contracts. In *2019 IEEE 23rd International Enterprise Distributed Object Computing Conference (EDOC)*, pages 77–87, 2019.

[FHBL19] Ghareeb Falazi, Michael Hahn, Uwe Breitenbücher, and Frank Leymann. Modeling and execution of blockchain-aware business processes. *SICS Software-Intensive Cyber-Physical Systems*, 34(2):105–116, 2019.

[FLB$^+$20] Ghareeb Falazi, Andrea Lamparelli, Uwe Breitenbuecher, Florian Daniel, and Frank Leymann. Unified integration of smart contracts through service orientation. *IEEE Software*, 37(5):60–66, 2020.

[Kne19] Markus Knecht. Mandala: A smart contract programming language. *CoRR*, abs/1911.11376, 2019.

[LFB$^+$20] Andrea Lamparelli, Ghareeb Falazi, Uwe Breitenbücher, Florian Daniel, and Frank Leymann. Smart contract locator (scl) and smart contract description language (scdl). In Sami Yangui, Athman Bouguettaya, Xiao Xue, Noura Faci, Walid Gaaloul, Qi Yu, Zhangbing Zhou, Nathalie Hernandez, and Elisa Y. Nakagawa, editors, *Service-Oriented Computing – ICSOC 2019 Workshops*, pages 195–210, Cham, 2020. Springer International Publishing.

[LTHT19] Jian-Wei Liao, Tsung-Ta Tsai, Chia-Kang He, and Chin-Wei Tien. Soliaudit: Smart contract vulnerability assessment based on machine learning and fuzz testing. In *2019 Sixth International Conference on Internet of Things: Systems, Management and Security (IOTSMS)*, pages 458–465, 2019.

[LWX$^+$19] Zixin Li, Haoran Wu, Jiehui Xu, Xingya Wang, Lingming Zhang, and Zhenyu Chen. Musc: A tool for mutation testing of ethereum smart contract. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1198–1201, 2019.

[LXL$^+$19] Qinghua Lu, Xiwei Xu, Yue Liu, Ingo Weber, Liming Zhu, and Weishan Zhang. ubaas: A unified blockchain as a service platform. *Future Generation Computer Systems*, 101:564–575, 2019.

[Nak19] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. Technical report, Manubot, 2019.

[OO14] Diego Ongaro and John Ousterhout. In search of an understandable consensus algorithm. In *2014 {USENIX} Annual Technical Conference ({USENIX}{ATC} 14)*, pages 305–319, 2014.

[QTGS21] Peng Qin, Weiming Tan, Jingzhi Guo, and Bingqing Shen. Intelligible description language contract (idlc)–a novel smart contract model. *Information Systems Frontiers*, pages 1–18, 2021.

[Sch01] 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*, pages 101–102. IEEE, 2001.

[Sen19] Martina Sengstschmid. *Community blockchain interaction patterns*. Vienna, 2019.

[SJS$^+$18] Stefan Schmidt, Marten Jung, Thomas Schmidt, Ingo Sterzinger, Günter Schmidt, Moritz Gomm, Klaus Tschirschke, Tapio Reisinger, Fabian Schlarb, Daniel Benkenstein, et al. Unibright-the unified framework for blockchain based business integration. *White paper, April*, 2018.

[SKH18] Ilya Sergey, Amrit Kumar, and Aquinas Hobor. Scilla: a smart contract intermediate-level language. *CoRR*, abs/1801.00687, 2018.

[SMG19] Sarwar Sayeed and Hector Marco-Gisbert. Assessing blockchain consensus and security mechanisms against the 51 *Applied Sciences*, 9(9), 2019.

[SRS19] Eder Scheid, Bruno Rodrigues, and Burkhard Stiller. Toward a policy-based blockchain agnostic framework. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 609–613, 2019.

[TYSS19] T. Tateishi, S. Yoshihama, N. Sato, and S. Saito. Automatic smart contract generation using controlled natural language and template. *IBM Journal of Research and Development*, 63(2/3):6:1–6:12, 2019.

[ZKCS20] Nejc Zupan, Prabhakaran Kasinathan, Jorge Cuellar, and Markus Sauer. *Secure Smart Contract Generation Based on Petri Nets*, pages 73–98. Springer Singapore, Singapore, 2020.

[ZSJB18] Muhammad Ahmad Zafar, Falak Sher, Muhammad Umar Janjua, and Salman Baset. Sol2js: Translating solidity contracts into javascript for hyperledger fabric. In *Proceedings of the 2nd Workshop on Scalable and Resilient Infrastructures for Distributed Ledgers*, SERIAL'18, page 19–24, New York, NY, USA, 2018. Association for Computing Machinery.