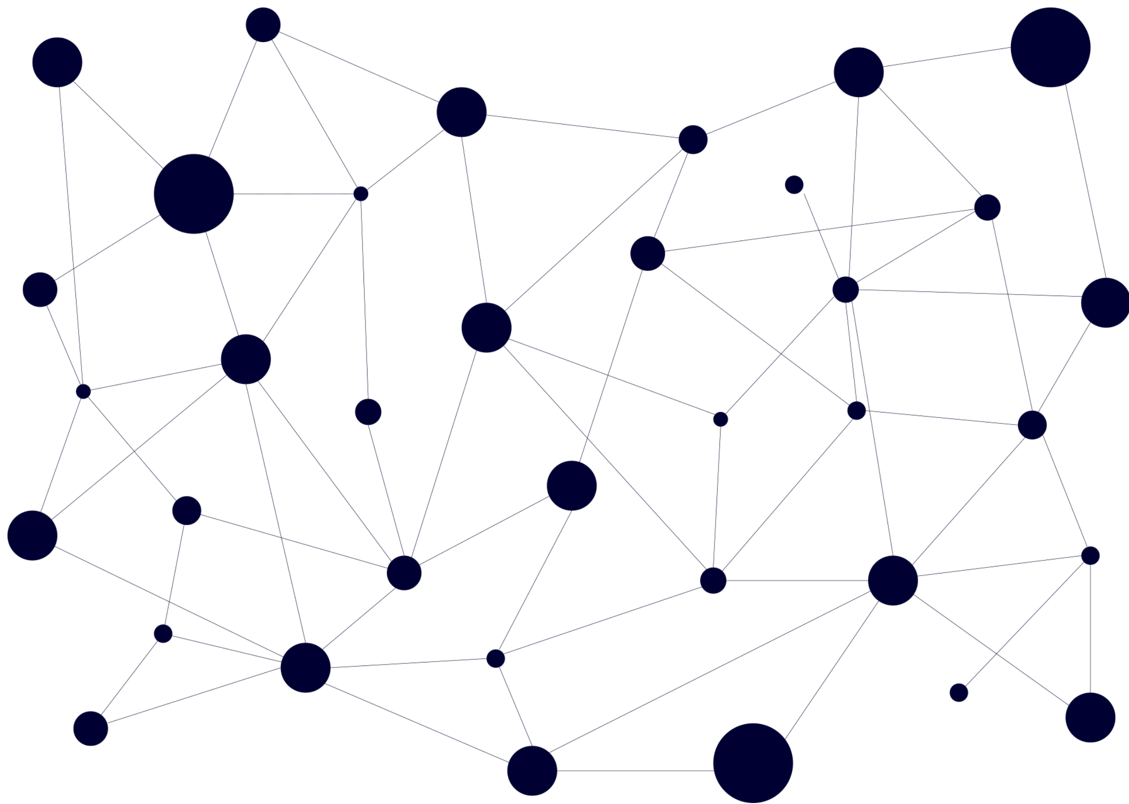


Comparative Analysis of Blockchain Technologies for Data Ownership and Smart Contract Negotiation

Master's Thesis in Computer Systems and Networks

VAIOS TAXIARCHIS
MALAMA KASANDA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

MASTER'S THESIS 2019

Comparative Analysis of Blockchain Technologies for Data Ownership and Smart Contract Negotiation

VAIOS TAXIARCHIS
MALAMA KASANDA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2019

"Comparative Analysis of Blockchain Technologies for Data Ownership and Smart Contract Negotiation"

VAIOS TAXIARCHIS
MALAMA KASANDA

© VAIOS TAXIARCHIS, 2019.
© MALAMA KASANDA, 2019.

Supervisor: Gerardo Schneider, Computer Science and Engineering
Examiner: Carl-Johan Seger, Computer Science and Engineering

Master's Thesis 2019
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2019

Abstract

Blockchain technologies have gained significant popularity in the recent past. They are distributed ledgers that record all the transactions on the network and allows parties who do not trust each other to transact. The parties collaborate in its maintenance. The Blockchain's distributed nature connotes multiple ethical and privacy concerns, such as unauthorized access and control of decentralized applications. In this thesis, we first survey the state of the art, focusing on a comparative analysis of public and private implementations of distributed ledgers. Additionally, we present a proof of concept (PoC) implementation on the access control to certain parts of smart contracts as well as investigate the possibility of negotiating the terms of a smart contract. We make use of the commercial real estate *leasing* operations as a case study. We conduct a comprehensive evaluation of our PoC. This evaluation demonstrates a trade-off in the choice of blockchain technology for building distributed applications. Drawing from the process flow of paper based contract negotiations, we design a library mechanism that can be used to negotiate contract terms in a bilateral fashion.

Keywords: blockchain, smart contract, access control, commercial real estate, Ethereum, Hyperledger Fabric

Acknowledgements

We would like to express our undying gratitude to our supervisor Professor Gerardo Schneider under whose guidance this work came to fruition. Furthermore, we would love to extend a vote of thanks to our examiner Professor Carl-Johan Seger whose constructive feedback went a long way during the course of our work.

Vaios Taxiarchis and Malama Kasanda, Gothenburg, 2019

Contents

List of Figures **xiii**

List of Tables **xv**

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Goals and Research Questions . . . 2
- 1.3 Limitations . . . 4
- 1.4 Methodology . . . 4
- 1.5 Outline . . . 5

2 Theoretical Background **7**
- 2.1 Digital Cryptography . . . 7
  - 2.1.1 Cryptographic Hash Functions . . . 8
  - 2.1.2 Symmetric-key Cryptography . . . 9
  - 2.1.3 Public-key Cryptography . . . 10
- 2.2 Blockchain Technology . . . 11
  - 2.2.1 Public Blockchain . . . 12
    - 2.2.1.1 Bitcoin: A Peer-to-Peer Electronic Cash System . . . 13
  - 2.2.2 Private Blockchain . . . 15
  - 2.2.3 Smart Contracts . . . 16
- 2.3 Use Case: Commercial Real Estate . . . 17
- 2.4 Role-Based Access Control . . . 18
- 2.5 Related Work . . . 19

3 Comparative Analysis **21**
- 3.1 Ethereum . . . 21
  - 3.1.1 Ethereum Virtual Machine . . . 21
  - 3.1.2 Ether and Gas . . . 22
  - 3.1.3 Accounts, Transactions and Messages . . . 23
    - 3.1.3.1 Accounts . . . 23
    - 3.1.3.2 Transactions . . . 23
    - 3.1.3.3 Messages . . . 23
  - 3.1.4 Consensus Algorithm (Mining) . . . 24
  - 3.1.5 Smart Contracts Deployment . . . 25
- 3.2 Hyperledger Fabric . . . 26
  - 3.2.1 Transaction Processing . . . 27

3.2.1.1 Endorsement Policies 28
3.2.1.2 Consensus Algorithm 28
3.2.2 Membership Services and Identity Management 29
3.3 Comparison 29
3.3.1 Similarities 29
3.3.2 Differences 30
3.3.2.1 Network operation 30
3.3.2.2 Consensus algorithm 31
3.3.2.3 Smart contracts 31
3.3.2.4 System currency 32
3.3.3 Conclusion 32

4 Implementation 33
4.1 Design and Specification 33
4.1.1 System Description 34
4.1.2 Design of the Smart Contract 35
4.1.3 Design of the Negotiation Mechanism 37
4.1.4 Role-based Access Control Model 38
4.2 Hyperledger Fabric Implementation 39
4.2.1 System Setup 40
4.2.2 Transaction Processing 41
4.2.2.1 Access Restriction 42
4.2.3 Negotiation Mechanism 43
4.3 Ethereum Implementation 43
4.3.1 System setup 44
4.3.1.1 Ethereum Node 44
4.3.1.2 Remix Web Browser IDE 44
4.3.2 Building the Smart Contract 45
4.3.2.1 State Variables 45
4.3.2.2 Functions 45
4.3.2.3 Events 46
4.3.2.4 Access Restriction (Modifiers) 47
4.3.2.5 Helper Functions 48
4.3.3 Negotiation Mechanism 49
4.3.3.1 Libraries in Solidity 49
4.3.3.2 Library Implementation 49
4.3.4 Wallet Recovery 50

5 Evaluation 53
5.1 Evaluation Criteria 53
5.2 Descriptive Evaluation 54
5.2.1 Leasing Operation 54
5.2.2 RBAC Mechanism 57
5.2.3 Negotiation Library 60
5.2.3.1 Wallet Recovery 61
5.3 Ethereum gas consumption 62

6 Conclusion **65**
6.1 Discussion 65
6.2 Future Work 66

Bibliography **69**

A Appendix A **I**

B Appendix B **IX**

List of Figures

2.1 Symmetric-key cryptography (single secret key). 9
2.2 Public-key cryptography (private and public key). 10
2.3 Blockchain as a linked list data structure. 11
2.4 How digital assets are transferred on the Bitcoin blockchain. 13
2.5 How Bitcoin solves the double spending problem. 14
2.6 Bank system vs. public blockchain vs. private blockchain. 15
2.7 Creation and execution of a smart contract in steps. 16

3.1 Ethereum transaction and gas consumption. 24
3.2 Ethereum smart contracts deployment process. 25
3.3 Transaction processing in Hyperledger Fabric. 27

4.1 The deployment and the signing of the smart contract. 35
4.2 The payment process and the transfer of digital money. 35
4.3 The termination of the agreement by the landlord. 36
4.4 The smart contract negotiation process initiated by the tenant. 37
4.5 Simplified CRE business network model in Hyperledger. 40
4.6 Interaction between the smart contract and the library. 50
4.7 Tenant links a new wallet using the private recovery key. 51

5.1 Smart contract deployment on Ethereum blokchain. 55
5.2 Negotiation library deployment on Ethereum blokchain. 55
5.3 Deployment of the business network to Hyperledger Fabric. 56
5.4 Creation of two network participants on Hyperledger Fabric. 56
5.5 The potential tenant reads the terms and signs the contract. 58
5.6 A non-authorized user attempts to sign the smart contract. 58
5.7 Participant identities issued by the administrator. 59
5.8 Tenant attempts to access a contract assigned to another tenant. . . . 59
5.9 Terms negotiation between the landlord and potential tenant. 60
5.10 Terms negotiation transaction in Hyperledger Fabric. 61
5.11 Landlord links a new wallet to the smart contract. 62
5.12 Ethereum price in relation to USD and Bitcoin [47]. 63

List of Tables

3.1 Differences between Ethereum and Hyperledger Fabric. 30

4.1 Smart contract basic operations for landlord and tenant. 36
4.2 Smart contract functions used for negotiations of the terms. 37
4.3 Access control list for the smart contract basic operations. 38
4.4 Access control list for the terms negotiation functions. 39

5.1 Evaluation criteria for Ethereum and Hyperledger Fabric. 54
5.2 Evaluation of the RBAC mechanism for the leasing operation. 57
5.3 Ethereum performance evaluation in terms of gas consumption. . . . 63

Listings

4.1 Sample transaction processor function to make payments. 41
4.2 Access Control List (ACL) rule to restrict access to the read terms function. 42
4.3 Example modifier used to check a condition prior to executing the function. 47

1

Introduction

1.1 Motivation

Blockchain technologies have in the recent past become popular, mainly due to the success of Bitcoin [1]. A blockchain is simply a decentralized peer-to-peer network, where each participant maintains a replica of a shared ledger of digitally signed transactions. Blockchains store records in groups called "blocks" and each of them is time-stamped and linked to the previous block using cryptographic functions [2]. It is a fault tolerant network that establishes trust using a distributed consensus algorithm. In addition, the information recorded on the blockchain is append-only using cryptographic means that guarantee the integrity of the recorded information. This immutability property makes it possible to establish the provenance of information.

First conceptualized in 2008 by Satoshi Nakamoto [1], as the technology to fuel the creation of Bitcoin, the blockchain's use has since evolved beyond cryptocurrencies to support Turing complete state machines [3]. For instance, Ethereum [4] and Hyperledger Fabric [5] support self-executing functions that can be stored on the blockchain network, known as *smart contracts*, and utilized to create decentralized applications (dApps). Interest from industry has given birth to a new type of blockchain called private (or *permissioned*) where participants need to be authenticated before joining the network as opposed to earlier blockchains where participants are free to join and leave the network as they please, these blockchains are called public (or *permissionless*). The provenance, immutability and finality features (See Chapter 2 for more on these features) make the blockchain suitable for several application use cases. Applications such as banking and insurance [12], digital identity and trade finance [13], supply chain business and commercial real estate [14], audit and compliance [15] have since been proposed or are under development.

Amid the digitization of information, people all around the world are increasingly becoming protective of their personal data. This protective nature of personal data has been exacerbated by recent media reports of data breaches in the form of the *Cambridge Analytica* [10] that entangled the worlds largest social media site Facebook. Alongside the data privacy revolution, Satoshi Nakamoto [1] defined an electronic cash system as a chain of digital signatures. Bitcoin's transactions are

tracked through the digital signatures that permanently reside in the blockchain. As a result, it is possible to infer the identity of a peer from this chain. While distributed ledgers provide cryptographic encryption to safeguard information, it lacks the flexibility of traditional systems when it comes to granting and removing access to data [16]. Consequently, business networks want to own their information so that they can only grant access to trusted members. However, data privacy and ownership are well-known concerns in the blockchain community. Thus, business houses looking to adopt the blockchain technology are highly interested in addressing these privacy issues, because their users want to control and own their data, such as transaction history [17].

Public blockchains operates with full transparency, allowing everyone to access the data stored in the distributed ledger, thus it guarantees only a low level of privacy to its users. On the other hand, private blockchains which use a central authority (CA) to verify their users cannot provide any transparency at all. Hence, there is an innate trade-off between transparency and privacy. A blockchain providing public verifiability of its overall state without revealing any information about the state of each participant [18], can guarantee privacy while the process of state transitions is made in a transparent way. In the case of smart contracts [4, 5], the owner of the contract needs to control a small amount of private, sensitive data like transaction history. One way of doing this is to implement a role or attribute-based access control to the smart contract [27, 28, 29]. Consequently, business networks that operate in an environment with competitors can limit the execution of parts of the smart contracts that reveal certain information deemed as either confidential or personal. For example, in a commercial real estate business, a lessee might have an exclusive agreement at a special rate with a lessor. The lessor might want to keep this contract private, and anyone not purview to it should not execute the function in the smart contract that reveals the amount involved.

In their current form, smart contracts are not sufficient to replace many of today's paper-based contracts, due to their lack of flexibility in terms of changing the terms of the contract [26]. Smart contracts operating on the blockchain can provide immutability, but once they are deployed, it is impossible to change any of their terms. As a result, it is immensely costly to run smart contracts in a volatile environment (where terms of an agreement may need to be re-adjusted as new factors creep in). Hence, there is a need for a mechanism that allows parties to propose and agree on new terms.

1.2 Goals and Research Questions

The aim of this thesis is twofold. Firstly, we will present a keyhole comparison between Ethereum [4] (public) and Hyperledger Fabric [5] (private), focusing on the architectural similarities and differences in relation to *data ownership* i.e specific parties have access to specific contents of the smart contract.

Secondly, we will design a mechanism for negotiating the terms of a smart contract involving two parties. Based on the previous comparison we will study the feasibility of implementing it in both kinds of blockchain networks. We further exemplify this comparative analysis by designing and implementing a Role-Based Access Control (RBAC) paradigm around a use case application to enhance data ownership.

Blockchains track ownership of transactions in a shared ledger based on the source and the destination ID of the transaction [1]. Our main research questions are:

- **What is the main difference between public and private blockchain regarding the access restriction to the contents of the smart contract?** Private blockchains provide an access control mechanism to restrict the access of the participants in a smart contract, while public blockchains are built to operate publicly without any restrictions. We will perform a comparative analysis of these two types of blockchains by investigating the operational differences (network operation, consensus algorithm, etc.).

- **Can we design a library to achieve negotiation of smart contract terms and evaluate the library using a use case implementation?** In order to design the negotiation library, we will summarize the methodology proposed by Scoca *et al* in [26]. The summary of this methodology will help answer this question.

- **Can we implement a role-based access control to smart contract contents based on the source and destination ID of the transaction?** In order to implement our role-based access control to smart contract contents, we will model and evaluate a proof of concept (PoC) implementation around roles of the source and destination ID of transactions. In addition, an extensive literature review will be done to compare how transactions and smart contracts are handled in Ethereum and Hyperledger Fabric. The comparison will help answer this question.

- **If so, what are the implications on the scalability of such an implementation in Ethereum, a public blockchain?** This research question seeks to answer the scalability in terms of *ether* gas exhaustion by transactions. We evaluate our implementation to make sure transactions complete their tasks before the gas runs out as well as investigate what happens when peers disappear and re-join the network.

Contract negotiation can be a lengthy and tedious process, but with the advent of blockchain technology, this process can be shortened. Our present research is therefore intended to make the following contribution:

- Ease the contract negotiation process using smart contracts by designing a *library* mechanism that can be used to negotiate the terms of the smart contract which involves two parties in a specified use case implementation.

1.3 Limitations

Other surveys of blockchain frameworks [3, 20] center around the performance of blockchain technologies, which is a crucial part of the choice of blockchain technology yet does not present the blockchain as a platform that enhances data ownership. Our work is limited to the analysis of the methods for restricting the execution of smart contract contents in terms of data ownership of user information. Furthermore, we concern ourselves with enforcing access control at the programing language level as opposed to applying it at the blockchain or application level. In particular, we concentrate on implementing access control using the underlying smart contracts.

We chose to concentrate on the programming language level due to time constraints. The smart contract implementation languages of both Ethereum and Hyperledger *Fabric* are relatively easy to bootstrap. For example, Ethereum has several different client implementations including `C++`, `Go`, `Python`, `Java`, and `Haskell`. These tools provide flexibility with how a node interacts with the Ethereum network. Additionally, the development of smart contracts within the Ethereum network is straightforward, due to its matured ecosystem. The same can be said about the Hyperledger community that provides an interactive console that developers can build and execute smart contracts in an efficient way.

1.4 Methodology

Hevner *et al.* [46] proposed a conceptual framework as a guideline for evaluating Design Science and Information System research. In order to perform an evaluation process they proposed that an implementation meets the following specific *attributes*; “functionality, performance, completeness, consistency, accuracy, reliability, usability”. Given the scope of the thesis and the novelty of the blockchain technology, we decided to evaluate our PoC implementation according to the *descriptive* evaluation method. This method is divided into two parts: informed argument and scenarios. During the first part of the process, we use information from the knowledge base (e.g. relevant research) to build a convincing argument for the artifact’s utility, while in the second, we construct detailed scenarios around the artifact to demonstrate its utility. In addition to that, we conducted a performance evaluation specifically for the Ethereum implementation, to illustrate the gas consumption of the smart contract deployment and use in real time scenarios.

1.5 Outline

To complete this thesis, we organize our work as follows:

- **Chapter 2: Theoretical Background.** We present a thorough technical background to lay the foundation for our work. In addition, we explain in detail our use case and the need for Role-based Access Control (RBAC).

- **Chapter 3: Comparative Analysis.** We shade more light on Ethereum and Hyperledger Fabric as two famous blockchain systems available today. We further present the similarities and differences between these two blockchain systems.

- **Chapter 4: Implementation.** We describe the methodology and the tools we used to implement our Proof of Concept (PoC). Further, we propose a library for smart contract terms negotiation.

- **Chapter 5: Evaluation.** We present the evaluation process of our implementation.

- **Chapter 6: Conclusion.** We conclude the thesis with our final thoughts and suggest areas for further future work.

2

Theoretical Background

In this chapter, we present the technical background to our work. We provide a summary background regarding modern digital cryptography, in order to ease the description of blockchain afterwards. We also explain the blockchain as a data structure and related concepts such as smart contracts, public and private blockchains. In addition, we give a description of our use case as well as a summary of role-based access control and the work done by others in relation to our thesis.

2.1 Digital Cryptography

Modern computer technology has revolutionized the ancient art of encoding messages to render them unreadable to anyone but the intended recipient. Conversely, cryptography has had a profound effect on how we use our information systems today. The process of secure communication between two or more parties in the presence of third parties (adversaries) is called cryptography. Digital cryptography plays a major role in many applications such as banking transaction cards, computer passwords or e-commerce transactions [30]. At first, cryptography was closely associated with the encryption, but during the past decade cryptography is based on solving comple mathematical problems. Modern cryptography studies the techniques that used to prevent adversaries from accessing private information in various aspects of information security such as data confidentiality, data integrity, authentication and non-repudiation [31].

Digital cryptography is most often associated with the process of converting any type of information (*plaintext*) into impenetrable text (*ciphertext*). This process is also known as encryption. The reverse process of converting the *ciphertext* into *plaintext* is called decryption. The set of algorithms that creates the encryption and decryption operations is called *cipher*, and is controlled by an algorithm and a “key” [30]. The key is known only to the parties participating in the communication and it is not revealed in any case to the rest of the parties. The key usually consists of a string of characters (which is secret) and is used to decrypt a ciphertext. Typically, a “cryptosystem” consists of three basic algorithms, one for generating the key, one for the encryption process and one for the decryption process respectively.

Traditionally, there are two kinds of cryptosystems used in digital cryptography: symmetric and asymmetric. In *symmetric* systems, the parties use the same secret key to encrypt and decrypt the messages. However, in *asymmetric* systems, the parties use a public key to encrypt the messages and a private key to decrypt them [30]. The asymmetric cryptography scheme enhances the security of communication between the parties, but the data manipulation becomes very slow, since they use keys of larger size, comparing to the ones generated in symmetric cryptosystems [31]. DES (Data Encryption Standard) is one of the most commonly used cryptosystems using symmetric cryptography, but it has been replaced by AES (Advanced Encryption Standard). On the other hand, RSA (Rivest-Shamir-Adleman) and ECC (Elliptic Curve Cryptography) are the most typical examples of asymmetric cryptosystems which are widely used for secure data transmission [32].

2.1.1 Cryptographic Hash Functions

Hash functions are mathematical algorithms that map data of arbitrary size to data of a fixed size. The output of a hash function is called message digest, and a hash function with identical inputs always produces the same output. Cryptographic hash functions belong to a special class of hash functions that are used in cryptography. Cryptographic hash functions have all of the properties of the hash functions, but they are designed in a such a way that is infeasible to invert (one-way functions). Specifically, it is computationally very hard for an adversary to find two different messages (brute-force attempt) that produce the same message digest (hash of the message). Thus, it is not impossible to break such a system, but it is infeasible to do so by any known practical means or it takes a substantial amount of time for an adversary to break it.

MD5 [33], an upgrade of the prior variant called MD4, is one of the most widely used hash functions, but it is proved to be broken in practice. MD5 produces a 128-bit hash value and its only use is as a checksum to verify data integrity because it suffers from extensive vulnerabilities [34]. The United States National Security Agency designed SHA-1 [35] (Secure Hash Algorithm 1), a cryptographic hash function that produces a 160-bit (equivalent to 20-byte) hash value for a given input. SHA-1 established as a U.S. Federal Information Processing Standard many widely used security applications and protocols, such as TLS and SSL, PGP, SSH, S/MIME, and IPsec rely on that. However, since 2005 it has not been considered completely secure and Microsoft [36], Google [37], Apple [38] and Mozilla [39] have all announced that their respective browsers will stop accepting SHA-1 SSL certificates by 2017. Because SHA-1 collision attacks have finally become practical by Stevens *et al.* [41] which forge two PDF documents with the same SHA-1 hash in roughly $2^{63.1}$ SHA-1 evaluations, two upgrades to SHA-1 have been proposed over the past decade namely SHA-2 and SHA-3 [40] respectively.

2.1.2 Symmetric-key Cryptography

Symmetric-key cryptography covers the encryption methods which use a single private key for both encryption and decryption. In order to describe how symmetric-key cryptography works, we will give an example of a *secret-key cryptosystem* as a cryptographic solution to the privacy problem arising from the communication between sender and receiver. Formally, a secret-key cryptosystem can be defined as a tuple $(\mathcal{M}, \mathcal{C}, \mathcal{K}, \mathcal{E}, \mathcal{D})$ with the following properties [30, 31]:

1. $\mathcal{M}$ is a set called the *message space* (the elements are called messages).
2. $\mathcal{C}$ is a set called the *ciphertext space* (the elements are called ciphertexts).
3. $\mathcal{K}$ is a set called the *key space* (the elements are called keys).
4. $\mathcal{E} = \{E_k : k \in \mathcal{K}\}$ is a set of functions $E_k : \mathcal{M} \to \mathcal{C}$ (each of them called *encryption function*).
5. $\mathcal{D} = \{D_k : k \in \mathcal{K}\}$ is a set of functions $D_k : \mathcal{C} \to \mathcal{M}$ (each of them called *decryption function*).
6. For each $e \in \mathcal{K}$, there is a $d \in \mathcal{K}$ such that $D_d(E_e(m)) = m$, for all $m \in \mathcal{M}$.

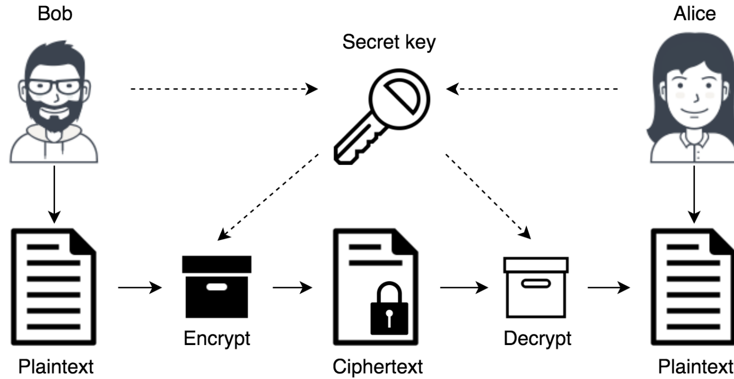


Figure 2.1: Symmetric-key cryptography (single secret key).

Figure 2.1 illustrates the communication between Alice and Bob over an insecure channel (in the presence of adversaries) using a symmetric-key cryptosystem. Before the initiation of any communication, Alice and Bob have to agree upon an encryption scheme: a message space $\mathcal{M}$, a ciphertext space $\mathcal{C}$, a key space $\mathcal{K}$ and two set of functions $\mathcal{E}$ and $\mathcal{D}$ for encryption and decryption respectively. The two parties have to agree upon a common secret key $k \in \mathcal{K}$ such that $D_k(E_k(m)) = m$. In this particular scenario, Bob wants to send a message to Alice. In order to do so, he encrypts his message $m \in \mathcal{M}$ using the encryption function $\mathcal{E}$ such that $c = E_k(m)$ and then sends it to Alice. Alice then decrypts the ciphertext c using the decryption function $\mathcal{D}$ and reads the message $m = D_k(c)$.

Symmetric-key cryptography has an evident problem since both of the parties have to obtain the secret key (shared). The key distribution requires communication over a secure channel, that in most cases is impractical. Whitfield Diffie and Martin Hellman have proposed a solution to the key distribution problem in 1976 [42] in which two different keys (public and private) are used.

2.1.3 Public-key Cryptography

In contrast to symmetric-key cryptography, public-key cryptography uses two different keys, the public key that is freely shared among the parties and the private key of each party that must remain secret (each party knows its own key). In public-key cryptosystems, the public (shared) key is used for encryption, while the private (secret) key is used for decryption. Formally, a *public-key encryption* (PKE) scheme can be defined as a triple of algorithms $(Gen, \mathcal{E}, \mathcal{D})$ with the following properties [43]:

1. The *key generation algorithm* $Gen(1^k)$ generates a triple of $(p_k, s_k, \mathcal{M}_k)$ where p_k is the public-key, s_k is the secret key, and $\mathcal{M}_k)$ is the message space associated with the p_k, s_k pair.

2. The *encryption algorithm* $\mathcal{E}$ takes as an input a message $m \in \mathcal{M}_k$ and outputs a ciphertext $c = \mathcal{E}_{p_k}(m)$.

3. The *decryption algorithm* $\mathcal{D}$ takes as an input a ciphertext c and outputs a message $m' = \mathcal{D}_{s_k}(c)$.

4. For every message $m \in \mathcal{M}_k$, $m' = m$ and $D_{s_k}(E_{p_k}(m)) = m$.

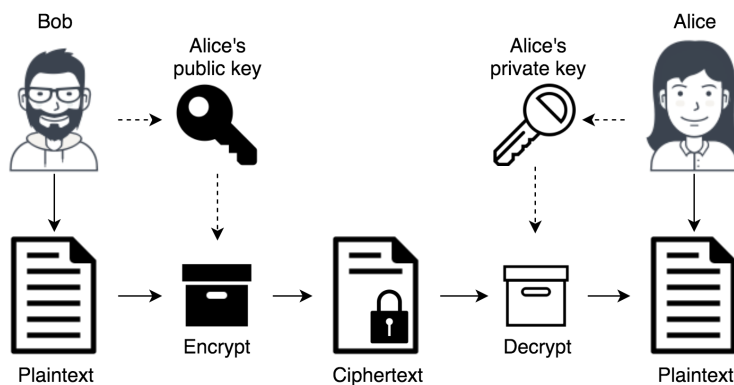


Figure 2.2: Public-key cryptography (private and public key).

Figure 2.2 illustrates the communication between Alice and Bob over an insecure channel using a public-key encryption scheme. Before the initiation of any communication, Alice and Bob have to agree upon an encryption transformation and each of them generate their keys pk_A, sk_A and pk_B, sk_B respectively, such that

$D_{sk}(E_{pk}(m)) = m$, for all $m \in \mathcal{M}_k$. The public keys are made public to both parties while the private keys remain secret. In this particular scenario Bob wants to send a message to Alice, thus he encrypts his message using Alice's public key pk_A and the encryption algorithm $\mathcal{E}$ such that $c = E_{pk_A}(m)$ and then sends it to Alice. Alice then decrypts the ciphertext c using her private key sk_A and the decryption function $\mathcal{D}$ to read the message $m = D_{sk_A}(c)$. The first public-key cryptosystem was proposed by Rivest, Shamir, and Adleman in 1978 [44] also known as the RSA algorithm. RSA is widely used for secure data transmission, and it is based on the practical difficulty of the factorization of the product of two large prime numbers (factoring problem).

2.2 Blockchain Technology

Blockchain technology is a mechanism that untrusted participants can share information by using a single digital history log (distributed ledger). A common digital history is important because digital assets and transactions are, in theory, easily faked or duplicated [1]. Blockchain technology solves this problem without using a trusted intermediary. At the heart of the blockchain are *blocks* linked together to form a *chain*. Figure 2.3 shows how these multiple blocks are connected to each other to form a chain (the blockchain).

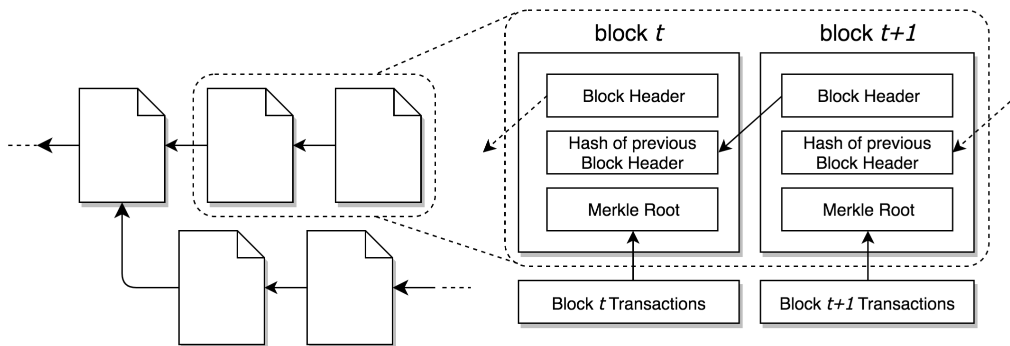


Figure 2.3: Blockchain as a linked list data structure.

Each block contains its own block header, the hash of the previous block and a collection of time-stamped valid transactions. The hash of the previous block links the blocks together, hence the blockchain can be thought of as a linked list data structure. Furthermore, the hash of the previous block cryptographically prevents the blocks from being altered or a block to be inserted between two existing blocks. As such, each subsequent block strengthens the verification of the previous block and hence the entire blockchain. Furthermore, a transaction can be any transfer of value between network participants (e.g. from one Bitcoin address to another). All of the previous transactions are hashed and paired, creating a single final hash value, called the Merkle root (from the Merkle tree).

Today, traditional methods of recording business transactions allow participants to maintain their own ledgers of transactions. This traditional method can be expensive, in some measure because third parties may charge fees for this service (e.g. renting cloud computing infrastructure for databases services). Clearly, this method is inefficient due to the duplication of efforts to maintain a number of ledgers for the same transactions. Furthermore, it may prove as a single point of failure if a central entity (for example a bank) is compromised, due to fraud or a cyber attack. On the other hand, the blockchain introduces efficiency into business networks by eliminating duplication efforts of recording transactions on numerous ledgers. Furthermore, transactions are secure and authenticated. As already noted, the blockchain technology has the following characteristics and advantages over traditional methods of recording transactions:

1. ***Provenance***: network participants are able to determine the origin of the asset and how its ownership has changed over time. This feature provides increased trust and no authority "owns" provenance of assets. In other words, blockchains can be described as *systems of proof* [5].

2. ***Consensus***: a mechanism to guarantee that the information added to the distributed ledger is valid (majority of the network nodes are in agreement). This feature prevents the double spending or other invalid data from being appended to the blockchain by ensuring that the next block being added represents the most current transactions on the network.

3. ***Immutability***: participants are not allowed to modify or delete a transaction once it has been committed to the ledger. To correct an erroneous transaction, a new one must be issued and both transactions will be visible on the ledger. This feature helps to lower the cost of audit and compliance.

4. ***Finality***: the single shared ledger becomes the only go-to place to verify the existence of an asset or completion of a transaction as opposed to checking various ledgers for the existence of transactions.

The blockchain technology provides business networks an opportunity to conduct their transactions in a secured, shared ledger that reduces the inefficiencies associated with traditional record systems, such as time and costs of maintaining separate ledgers while improving trust among participants and keeping all transactions visible.

2.2.1 Public Blockchain

Public blockchain networks are entirely open, and anyone can join and participate in the system. Strictly speaking, blockchain networks can be *public* in two ways:

- *Anyone* can **write** data **to** the blockchain.

- *Anyone* can **read** data **from** the blockchain.

The network usually has incentives to encourage more participants to join and to discourage network participants from cheating. Mining is the mechanism that enables decentralized security on the blockchain [15]. Miners validate new transactions and record them on the global ledger by solving intricate mathematical problems based on a cryptographic hash algorithm. Miners receive a reward whenever they "solve" a block (transactions contained in the block are considered confirmed). Bitcoin [1] is the typical example of a public blockchain network today. The openness and transparency of public blockchain come with a cost, which implies no privacy for the transactions.

The main advantage of public blockchains is that they have plenty of use cases due to their "openness". However, one of the disadvantages of public blockchains is that they require a substantial amount of computational power to maintain a very large distributed ledger. In other words, every node in the network has to solve a resource-intensive cryptographic problem to reach consensus.

2.2.1.1 Bitcoin: A Peer-to-Peer Electronic Cash System

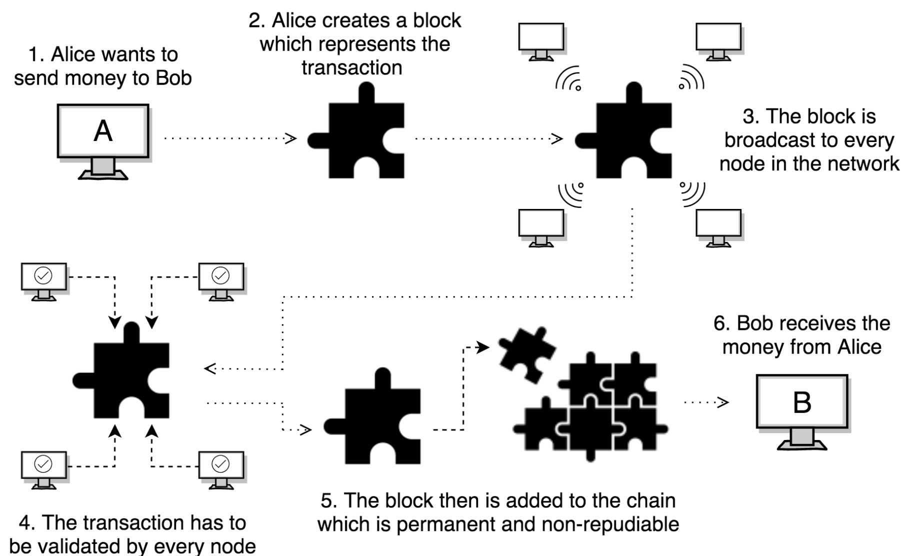


Figure 2.4: How digital assets are transferred on the Bitcoin blockchain.

As earlier stated, Satoshi Nakamoto introduced Bitcoin as an electronic payment system combining the concepts of cryptography and distributed systems. The main goal of Bitcoin is to allow the transfer of digital assets without the need for a trusted third party such as a bank. The Bitcoin network is a public blockchain that consists of nodes that verify and group transactions by solving cryptographic problems. Bitcoin users digitally sign their transactions using the hash of the previous transaction and the public key of the recipient. In other words, to send Bitcoins, the public key

of the recipient must be known beforehand. In a sense, the public key allows users to be pseudonymous.

Bitcoin users can come and go on the network and can create as many accounts as they wish. However, they need to accept the proof-of-work chain as evidence of the activities that happened in their absence. The transaction is broadcast across the network and is propagated to all peers in order for the recipient to verify the chain of ownership of the coin. Figure 2.4 illustrates how digital assets can be transferred on the Bitcoin blockchain [1]. Ideally, network participants bundle their transactions into a block, which is later broadcast across the network to all participants. Using a consensus algorithm called Proof of Work (PoW), the transaction is validated and verified by the network peers. Thereafter, the block is appended to the chain and the transaction is completed successfully.

Bitcoins are sent from a personal *wallet* - which is a client application used to generate transactions. To avoid the double-spending problem; a scenario where a payee sends the same coin value to different recipients, the system implements the longest acceptable chain available for that particular coin. The longest chain is a universal confirmation mechanism on the blockchain and is determined by the earlier accepted signatures of a coin. Figure 2.5 shows a *Buyer* that signs and sends a coin value to *Merchant A*, in the hope of tricking the system, the *Buyer* signs and sends the same coin value to *Merchant B* on a different address. Both transactions end up in the unconfirmed pool of transactions. However, only the first transaction was verified by the miners and added to the next block while the second transaction got fewer confirmations and hence was deemed invalid and pulled from the network rendering the transaction to be declined.

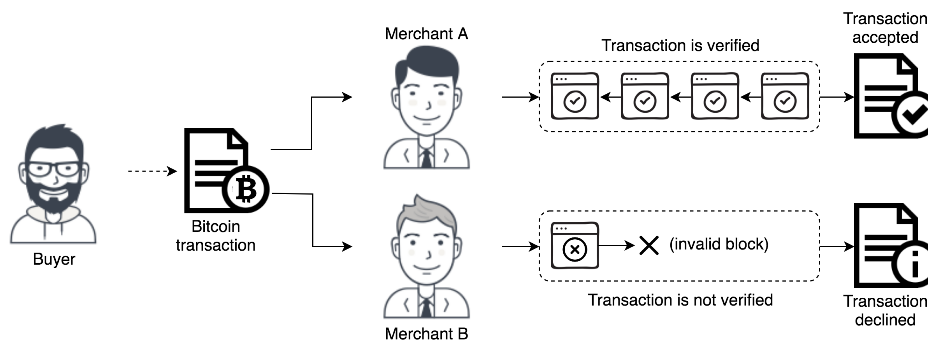


Figure 2.5: How Bitcoin solves the double spending problem.

The economics of Bitcoin allow the ecosystem to flourish. The generation of the *genesis* block introduces the first coin into the system. Subsequent block generation incentives allow a constant supply of Bitcoins into circulation that are owned to the block creators. At the time of writing this thesis, there was more than 130 billion USD worth of Bitcoins in existence, but Bitcoins will stop being created when the total number of coins reaches 21 billion in the year 2040 [7]. However, incentives can be realized from transaction fees as well. The difference between the output and

input value of a coin is the transaction fee which is added to the block containing the transaction. The block creator is paid this transaction fee as an incentive.

2.2.2 Private Blockchain

Conversely, in a *private* blockchain, network members are known and trusted to participate in the network. Strictly speaking, blockchains can be *private* in two ways:

- **write** permissions are restricted to authenticated members
- **read** permissions are either restricted or public

These permission-based networks restrict the members allowed to participate in the network, and as a consequence, the validation of transactions is delegated to special peers through a consensus algorithm. The participant receives an invitation to gain access to the network and contribute to the maintenance of the blockchain. Classic examples of this blockchain are Hyperledger Fabric [5] and Ripple [6].

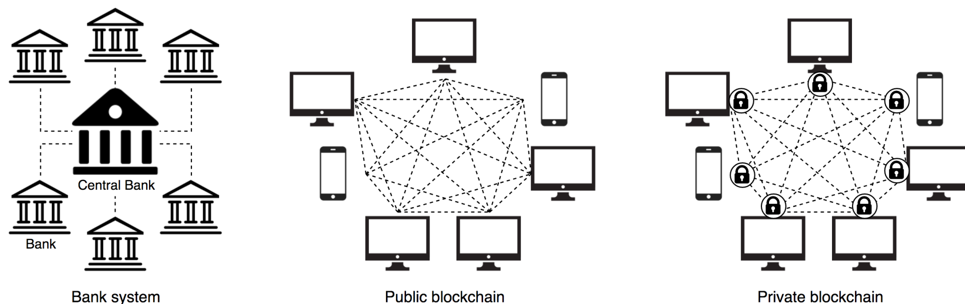


Figure 2.6: Bank system vs. public blockchain vs. private blockchain.

Figure 2.6 illustrates the key difference between the current bank system and the public and private blockchain. Generally speaking, private blockchains are usually administered by an organization or a trusted consortium that controls access permissions to the blockchain such as rights to read and modify the blockchain state. For the most part, these blockchains have received a lot of attention from financial institutions as they move to adopt the blockchain technology. This attention is attributed to the significant advantages private blockchains posses in comparison to their public counterparts. These advantages include the following:

- The organization whose jurisdiction the blockchain falls under can manage the rules of participation if the need arises. Furthermore, if **read** access is restricted, private blockchains can offer increased levels of privacy to the blockchain contents.
- As opposed to public blockchains, where transaction validation is done by a

number of nodes, in private blockchains transaction validators, are known, hence transactions are cheaper and block creation time is minimized. This advantage also prevents validators from colluding among themselves.

2.2.3 Smart Contracts

Nick Szabo [11] first defined a *smart contract* in 1994 as a "set of promises specified in digital form, including protocols that the parties perform to honor the promises". Figure 2.7 shows how smart contracts are created and executed on the blockchain. Szabo defined four design objectives for contracts which include:

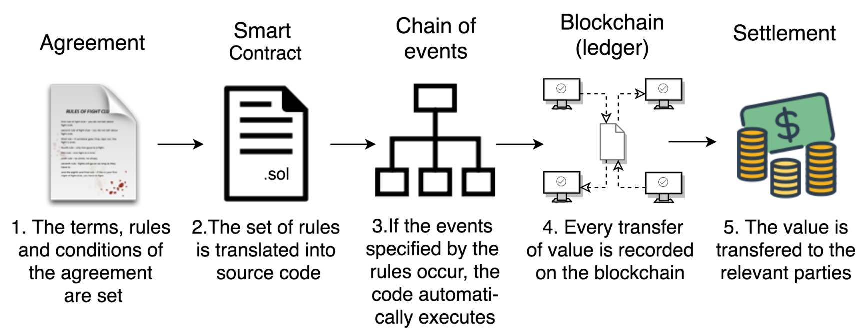


Figure 2.7: Creation and execution of a smart contract in steps.

- ***Observability***: the means for participating parties to observe each others' performance in regard to the contract or to prove their performance to each other. Ideally, the accounts department would see to it that the contracts an organization is involved with becoming more observable.

- ***Verifiability***: this is the ability for a party to prove to a body officially appointed to settle a dispute that the contract was honored or otherwise a dispute occurred.

- ***Privity***: this is the ability to distribute the knowledge or contents of the contract to parties that are only purview to it. In common law, third parties other than designated arbitrators or intermediaries should not have access to a contract.

- ***Enforceability***: these are means of enforcing the contract. One way is to institute *self-enforcing* schemes where the contract is executed upon a pre-set condition.

It can be noticed that the design objectives form two sets of observations; privity exerts restrictions over the contract contents, minimizing the exposure of the contract to third parties. On the other hand, observability, verifiability, and enforceability entail access to contractual information by third parties. Consequently, a trade-off must be met where controlled access to the contract is met while it is

possible to verify, enforce and observe the contract by third parties. These design properties enable both parties to observe the performance of the other party and verify if and when a contract has been performed, guarantee that only the details necessary for completion of the contract are revealed to both parties and be self-enforcing to eliminate the time spent observing the contract.

In the context of the blockchain, a smart contract is an agreement that governs business rules and provides business logic. It is executed automatically as part of the transaction and is stored on the blockchain. The state of the blockchain is changed by execution of the smart contract. Often, smart contracts allow the exchange of money, property, shares, or any asset in a transparent and conflict-free way. In most blockchain implementations, smart contracts have their own address or in some cases contain crypto-balance (e.g. Ethereum) and can be set up in three phases:

- Smart contract construction with agreed upon terms using a supported programming language.
- Deployment of the smart contract to the peers of the blockchain.
- Self-execution upon the occurrence of the pre-set condition. If however, the condition is external to the blockchain, a third party entity is needed to enter the information required for the smart contract to run.

The execution of the smart contract through the blockchain renders it reliable. The reliability is guaranteed using the consensus algorithm of the underlying blockchain. The blockchain network is able to verify whether the smart contract has been executed properly and resolve any dispute that may arise.

2.3 Use Case: Commercial Real Estate

Commercial Real Estate (CRE) denotes any property owned solely for the purpose of generating an income. CRE traditionally keeps several of its operations secret, such as comparable lease rentals, property prices, and valuations to create a competitive advantage. However, the inherent advantages of blockchains such as a distributed ledger, tamper-proof, censorship resistance, and smart contracts have the potential to transform CRE operations such as property purchase, financing, leasing, and management. To be specific, leasing, purchase, and sale transaction processes can benefit from blockchain adoption as it can take advantage of the benefits of the technology. The following illustrates how blockchain can promote CRE leasing, purchase and sales operations:

1. ***Property Searching***: The lessor and the lessee or their respective brokers list their requirements on a listing service. A transparent listing service system enables all parties to view the available listings based on their requirements. The property searching can be powered by a blockchain enabled listing service.

2. ***Pre-lease Due Diligence***: The lessor conducts a background check on the lessee, and the lessee checks the prior transactions and legal claims on the property.

3. ***Lease Agreement***: The key terms of the agreement are recorded on the blockchain and executed using a smart contract. The smart contract initiates payment of security deposit/advance rent either through cryptocurrency wallets or bank accounts using a payment interface.

4. ***Automated payments and cash flow management***: Based on the terms of the agreement, the smart contract initiates the regular lease payments from the lessee to the lessor, after paying the outstanding maintenance expenses to the contractors. The smart contract initiates the transfer of the security deposit to the lessor using the preferred mode of payment on completion of the lease term.

This use case provides us with an opportunity to note that data ownership is paramount in situations that comprise competitors. Once an agreement has been reached, participants should have the possibility to retain access control to their information. We will make use of the CRE operations to conclude our comparative study.

2.4 Role-Based Access Control

Role-Based Access Control (RBAC) is a common approach to manage users' access to resources or operations. RBAC enables the creation of hierarchies of roles and permissions [27, 28]. Permissions specify exactly which resources and actions can be accessed by whom. In our implementation, they can be realized in two ways [22]:

1. Using the inbuilt blockchain crypto-conditions,

2. Enforcing the access control through a third party-system like a cloud platform and build an asset transfer application on the blockchain.

In the first option, we can use property owners as partial owners of the property; primarily we can retain the ownership to the underlying blockchain in the data model and assign roles as lessee and lessor. As a result, a crypto-condition such as a threshold condition (1 of n) can be used to transfer assets. Basically, in a threshold condition, the application is modeled around assets, inputs, and outputs as a mechanism by which control of an asset is transferred. The amounts of an asset are encoded in the outputs of a transaction, and each output may be spent separately. To spend an output, the output's condition must be met by an input that provides a corresponding fulfillment. The second option is somewhat easier; we can model the applications access rights and allow a third party application to host them. However, there remain some challenges with these methods, which have been

addressed in previous research [22]. The first option does not scale easily because whenever partial players in the leasing business are added or removed, a new transfer trail of assets is created. The second option drifts away from the decentralized nature of the blockchain and creates a single point of failure in the third party application to host the access rights.

2.5 Related Work

This section discusses a number of implementations related to our thesis. It also describes how our thesis work differs from the proposed systems in the related work.

Public key infrastructures (PKI) facilitate the secure electronic transfer of information across various network applications (e.g. e-commerce, internet banking, confidential email). Well-known PKI-based systems, such as the most widely used email encryption standard, OpenPGP [23], and its implementation GnuPG [24], are complex systems with a very high cost of maintenance. Additionally, those systems rely on a Certificate Authority (CA), a trusted third party, which is responsible for the distribution and management of the digital certificates. The CA centralization creates a single point of failure, making both systems target for multiple attacks. A typical example is the DigiNotar [25] attack, where an attacker penetrated the Dutch CA DigiNotar and gained complete access to all eight of the company's certificate-issuing servers. In our implementation, we aim to investigate how to restrict access to smart contracts in a new efficient and decentralized way without the use of an additional central entity responsible for that.

Scoca *et al.* [26] presented a methodology for the autonomous negotiation of smart contracts in cloud services, by introducing a formal language to describe the interactions between offers and requests. They also analyzed the cost and the modifications required to reach consensus, by providing a full evaluation process. In our thesis work, we propose an implementation that allows dynamic inclusion of new terms within a smart contract. Flexible smart contracts that allow alteration in their contents by preserving privacy at the same time can be very useful in business-to-business networks. Particularly, we propose a library that can be used by any smart contract in order to handle negotiations between the involved parties in our use case scenario (landlord and tenant).

Ouaddah *et al.* [27] proposed a novel framework for access control in IoT-based on the blockchain technology. They proposed *FairAccess*, a fully decentralized pseudonymous and privacy-preserving authorization management framework that enables users to own and control their data. They also provide a reference model for the proposed framework within the objectives, models, architecture and mechanism specification in IoT. Likewise, Zhang *et al.* [28] proposed a smart contract-based framework, which consists of multiple access control contracts (ACCs), one judge contract (JC) and one register contract (RC), to achieve distributed and trustworthy access control for IoT systems. Each ACC provides one access control method for

a subject-object pair and implements both static access right validation based on predefined policies and dynamic access right validation by checking the behavior of the subject. In addition, the JC implements a misbehavior-judging method to facilitate the dynamic validation of the ACCs by receiving misbehavior reports from the ACCs, judging the misbehavior and returning the corresponding penalty. Both implementations focus on IoT security allowing their users to control their own data, while our implementation describes the design and implementation of a role-based mechanism in smart contracts for the purpose of real estate leasing operations.

Cruz *et al.* [29] proposed a blockchain-based RBAC system using the Bitcoin network as an infrastructure, aiming to provide an irrefutable proof of the role of a user (issued by an organization) by verifying the connection of the user to the organization through the Bitcoin blockchain. Specifically, whenever an unknown user claims to have a role from a particular organization, the system will create a Bitcoin transaction, then a service-providing organization will verify the Bitcoin transaction containing the addresses of the organization and the user. This implementation is inspired by the mechanism of Hyperledger Fabric, used for issuing and verifying user transactions from the rest of the users. Our thesis work aims to investigate how an RBAC scheme can be implemented in Ethereum blockchain, in order to create smart contracts that have the ability to restrict access to certain users. Hypeledger Fabric has a built-in mechanism to support access control in smart contracts, due to its permissioned type of operation.

3

Comparative Analysis

Since the inception of Bitcoin, there has been a surge in the development of industry grade blockchains. Although Ethereum and Hyperledger Fabric are some of the matured blockchain ecosystems on the market, their implementation styles differ significantly. In this chapter, we shade more light on the architecture of Ethereum and Hyperledger Fabric. Furthermore, we present a keyhole comparison between Ethereum and Hyperledger Fabric by analyzing their key similarities and differences.

3.1 Ethereum

Ethereum [4] was proposed towards the end of 2013 by Vitalik Buterin and officially launched in July of 2015. At the time of writing, Ethereum had a market capitalization of 60 billion USD, not counting its attraction as a platform of choice for a lot of decentralized applications as well as for the creation and launch of ICOs (Initial Coin Offerings). Ethereum is an open source platform, operating on a public blockchain network, providing easy, fast and reliable payments similarly to Bitcoin. But in addition to that, it enables developers to build and deploy decentralized applications, called smart contracts.

Ethereum smart contracts define rules and penalties for an agreement between members by enforcing those obligations [4]. Therefore, smart contracts are stored in the Ethereum blockchain as decentralized applications (dApps) and can used by the Ethereum users in the future. Ethereum allows developers to create smart contracts consisting of an unlimited number of operations, without restricting them in a set of limited operations [29].

3.1.1 Ethereum Virtual Machine

The Ethereum Virtual Machine (EVM) can be viewed as a distributed global computer where all smart contracts are executed [4]. In order to restrict the resources used by the smart contracts running in the EVM, every computation is paid in *ether* (ETH), the built-in currency of the Ethereum blockchain. Thus, smart contract op-

erations that is executed in the EVM, are broadcast to every node in the network. It is evident that the Ethereum project was built with the prospect of introducing such a sandboxed environment to simplify the smart contract development in the future. The EVM can be considered to be a “learning environment”, which gives developers a testing platform in order to ease the development process of applications on the blockchain. The EVM is a good testing bed because it is completely isolated from the rest of the main network. Additionally, the EVM has been implemented in `C++`, `Python`, `Ruby`, and a few other programming languages, while every Ethereum node in the network runs their own EVM implementation and is capable of executing the same instructions.

3.1.2 Ether and Gas

The native value token of Ethereum blockchain is called *ether* [4] and it is the main element for operating the distributed applications in Ethereum platform. Essentially, *ether* is the fuel of the Ethereum network, a form of payment made by the clients of the platform to the machines executing the requested operations. Thus, developers are motivated to write quality code to keep the network healthy, since the deployment of wasteful code leads to additional cost. Whenever a contract is executed as a result of being triggered by a message or transaction, every instruction is executed on every node of the network. Every operation executed on the blockchain comes with a cost, measured in *gas* units. *Gas price* is the amount of *ether* the sender is willing to spend on every unit of gas, and is measured in `gwei`. ‘`Wei`’ is the smallest unit of *ether*, where 1^{18} `wei` represents 1 *ether*. One `gwei` is 1,000,000,000 `wei`. With every transaction, a sender sets a *gas limit* and *gas price*. The product of *gas price* and *gas limit* represents the maximum amount of `wei` (or ‘*TX fee*’) that the sender is willing to pay for executing a transaction. Sending tokens will typically cost about 50000 `gwei` to 100000 `gwei`, so the total `TX fee` is about 0.001 to 0.002 of *ether*. This restriction provides an efficient way to reach consensus on the system without the need of trusted third parties, or intermediates.

Each smart contract execution is redundantly replicated across many nodes making the execution expensive. Therefore, Ethereum platform encourages only the execution of the necessary operations on the blockchain and all of the unnecessary operations can be executed offline (without using the blockchain itself). The miners can purchase gas for ether after they have executed the code of a smart contract. Due to the price fluctuation of ether as a result of the open market, the price of gas varies accordingly. Basically, the price of gas is decided by the miners, who can refuse to process a transaction with a lower gas price than their minimum limit. There is an automated process that purchases gas for ether in each Ethereum client according to the limit specified for the transaction. Moreover, Ethereum’s execution fees apply for every computational step within a smart contract, preventing intentional attacks or abuse on the Ethereum network.

3.1.3 Accounts, Transactions and Messages

3.1.3.1 Accounts

Transactions in Ethereum smart contracts can simply be considered as a transfer of *ether* from one Ethereum account to another. Generally, there are two types of accounts: Contract Accounts (CA) and Externally Owned Accounts (EOA). Contract accounts have an ether balance, an associated code and their code execution is triggered by transactions or messages (function call) received from other contracts. On the other hand, Externally Owned Accounts (EOAs) have an ether balance and they can send transactions by either simply transferring ether or trigger the code of a contract. An EOA is completely controlled by a private key and has no associated code.

3.1.3.2 Transactions

Transactions in Ethereum formally refer to the signed data package that stores a message that has to be sent from an Externally Owned Account (EOA) to another account on the blockchain. The transaction contains the Ethereum address of the recipient, a signature that identifies the sender, the amount of ether being transferred (`VALUE`), two extra values `STARTGAS` and `GASPRICE`, and an optional data field. The signature is used to identify the sender and verify their intention to send the message to the recipient using the Ethereum network. The `VALUE` field stores the amount of wei to transfer from the sender to the recipient, the `STARTGAS` field which represents the maximum number of computational steps the transaction execution is allowed to take, and the `GASPRICE` field represents the fee the sender is willing to pay for gas; one unit of gas corresponds to the execution of one atomic instruction (computational step).

3.1.3.3 Messages

Ethereum smart contracts can also send "messages" to other smart contracts. These messages are virtual objects that exist only in the Ethereum execution environment and can be used as function calls. Similarly, each message contains the identity of the sender and the recipient, the amount of ether being transferred (`VALUE`), an extra value `STARTGAS` and an optional data field. `STARTGAS` value restricts the amount of gas that can be consumed by the code execution triggering. Ethereum messages act like transactions and can only be produced by contracts, not from external accounts.

3.1.4 Consensus Algorithm (Mining)

Figure 3.1 illustrates the execution of an Ethereum transaction showing the gas consumption of every instruction in the transaction. Ethereum, like all blockchain technologies, uses a consensus algorithm which is based on choosing the block with the highest total difficulty [4]. In Ethereum, the difficulty is an integer that indicates how difficult it is for the miners to mine a new block by finding a hash below a given target, while total difficulty holds for the difficulty of the whole chain until this block. During this process, miners produce blocks which then other miners have to check for their validity. In particular, Ethereum uses a Proof-of-Work (PoW) system where all miners use a special software to solve mathematical problems.

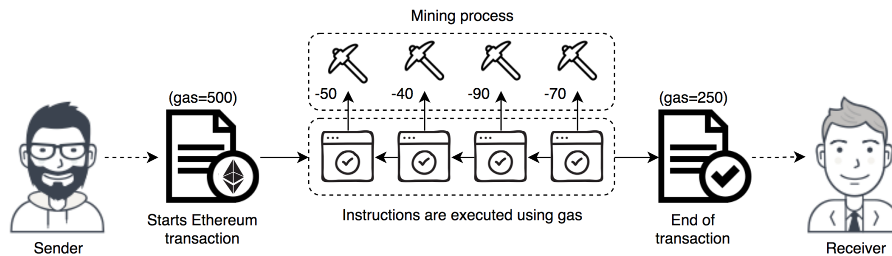


Figure 3.1: Ethereum transaction and gas consumption.

Generally, Ethereum blockchain is very similar to the Bitcoin blockchain, although Ethereum proposes a more enhanced consensus mechanism. The blocks of both blockchain implementations contain a copy of the transaction list and the most recent state, which is technically the root hash of the Merkle tree. In addition to that, each Ethereum block contains the number of the block and its difficulty. These two additional values give an extra level of security and the validation of the blocks becomes even harder. The PoW algorithm used in Ethereum platform is called *Ethash*, which is a modified version of the Dagger-Hashimoto algorithm [4]. This algorithm suggests that miners have to scan and test for a nonce to find a solution that is below a certain difficulty threshold. Specifically, the protocol indicates that the difficulty can be adjusted dynamically in such a way that on average one block is produced by the entire network every 15 seconds. Thus, any node participating in the Ethereum network can become a miner and their expected total reward from mining will be directly proportional to their mining power, or *hashrate*.

Moreover, *Ethash* is a memory intensive computational problem, making it application-specific integrated circuit (ASIC) resistant, thus, it allows a more equally distributed (decentralized) notion of security. In order to modify a block, someone has to redo the work spent on this block, including the work spent on the blocks that have been chained to it. Thus, the majority of the total computation power of the miners participating in the Ethereum network are controlled only by honest miners, enhancing the levels of security throughout the blockchain.

3.1.5 Smart Contracts Deployment

Ethereum offers a very large distribution of software tools for smart contract development. Figure 3.2 illustrates the steps of deploying a smart contract in Ethereum platform. Developers interested in the development of Ethereum smart contracts use the following steps to successively construct and deploy a smart contract to the blockchain system:

1. Write the source code of the smart contract in a file (or a group of files) with the extension `*.sol`, using the programming language Solidity.

2. Compile the file (or files) containing the source code of the smart contract using the Solidity compiler (`solc`). Developers can call the Solidity compiler, which runs inside the Ethereum node that they have already started, using the JavaScript API called `web3.js`. The correctness of the smart contracts can be verified in a private Ethereum network using a local `testnet`, without consuming any gas.

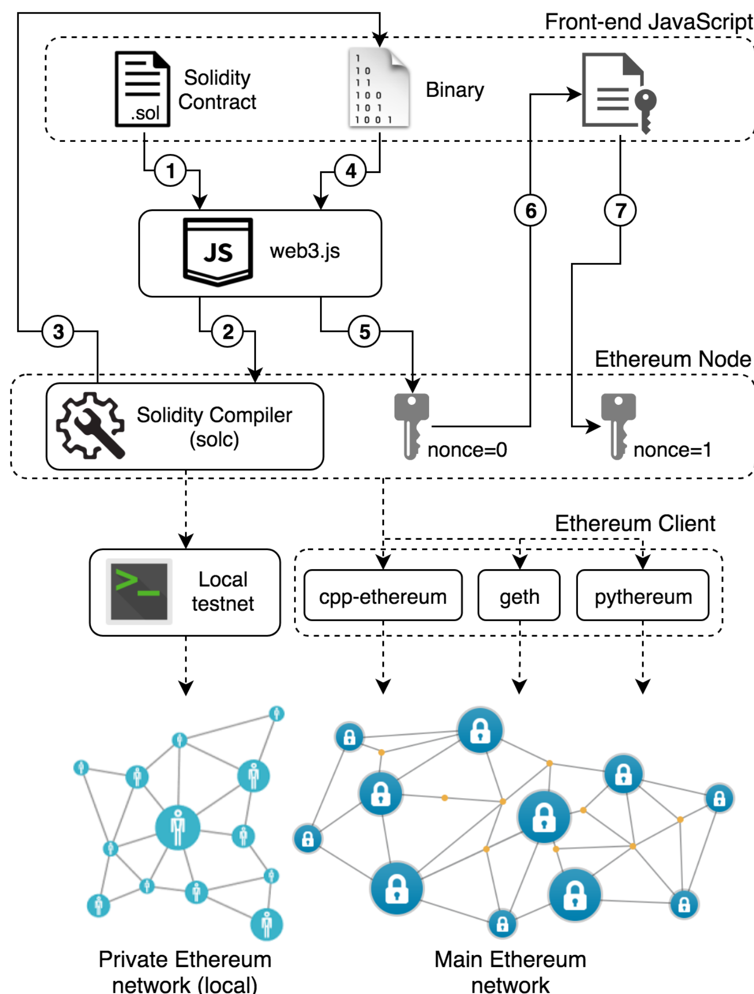


Figure 3.2: Ethereum smart contracts deployment process.

3. The binary file generated from the source file (or files) written in Solidity, is sent back to the dApp in the front-end JavaScript environment. In the Ethereum community, dApps provide a practical UI to interact with the smart contracts.

4. The dApp publishes the smart contract (the binary file of the smart contract) to the main network, using the `web3.js` JavaScript API.

5. The smart contract is signed using the Ethereum node default wallet address (or another Ethereum address). The step of deploying the smart contract to the network costs ether.

6. The Ethereum node sends back the blockchain address of the smart contract and the ABI (a JSON file containing all the variables, events and methods of the compiled smart contract).

7. Whenever the dApp calls a method of the published smart contract, it uses the blockchain address and ABI of the smart contract, along with the nonce value. Nonces start with a value equal to zero when the smart contract is published on the blockchain and they are used to prevent pushing duplicated transactions, while at the same time they increase the lifetime of the key.

3.2 Hyperledger Fabric

Hyperledger Fabric is a project housed under the Linux Foundation to provide industry grade private blockchain technology that renders immediate finality. At the time of writing this thesis, v1.1.0 was the current version in production. Hyperledger Fabric's smart contracts are called *chaincode* and written in either `Java` or `Go` programming languages and deployed to peers. Hyperledger Fabric is divided into two sets of peers: *validating* and *non-validating* peers. A validating peer is one that participates in the maintenance of the ledger, runs the consensus algorithm and validates transactions. On the contrary, non-validating peers *only* issue transactions to validating peers on behalf of clients. The validating peers execute transactions in the form of three transaction types to interact with the blockchain:

- *Deploy transaction:* This transaction installs the smart contracts to the peers.

- *Invoke transaction:* A particular chaincode that has been installed using the deploy transaction can be invoked with arguments specific to the type of transaction. The chaincode either writes or reads entries in its state and indicates whether it has failed or succeeded.

- *Query transaction:* This transaction reads and returns the state of the peers' permanent storage.

3.2.1 Transaction Processing

Unlike other blockchain technologies, Hyperledger Fabric takes a different approach to executing transactions. The system is designed to separate the execution of the smart contract and updating the distributed ledger [8]. A transaction between two clients with corresponding peers on the blockchain network makes use of the three transaction types listed above. Figure 3.3 shows the flow of the transaction to complete the asset exchange between two clients. The flow typically follows the steps listed below:

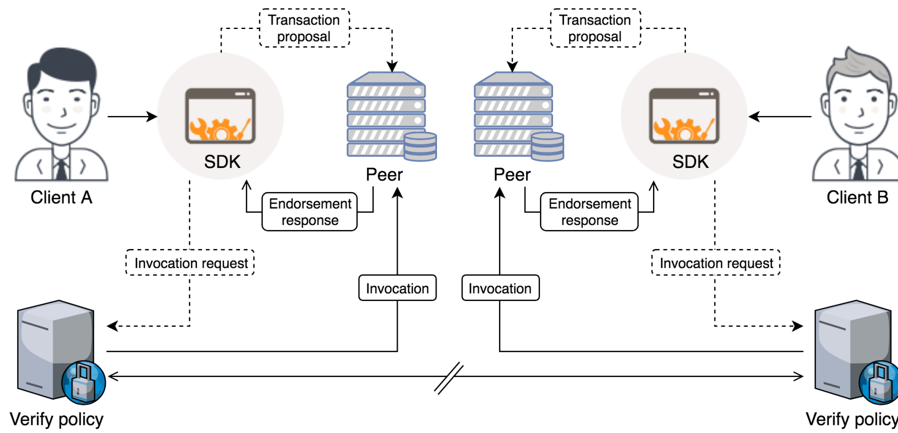


Figure 3.3: Transaction processing in Hyperledger Fabric.

1. *Client transaction initialization:* The client intending to partake in the transaction (e.g rent a house from a landlord) sends a request. This transaction request targets the corresponding peers on the blockchain network. Depending on the *endorsement policy* (See Section 3.2.1.1 for more) of that network, the transaction is directed to the right peers to sign the request. The transaction request is bundled into a transaction proposal by the Standard Development Kit (SDK) responsible for the smart contract. The transaction proposal marks the request to execute a smart contract function that reads/writes data to/from the ledger.

2. *Transaction endorsement*: This process takes care of the housekeeping duties before a transaction is committed to the ledger. In particular, the endorsing peers verify that the transaction is correctly issued and that it has not been submitted before. Furthermore, the endorsing peers make sure that the signature from *step 1* is valid and that the peer submitting the transaction has the necessary rights to do so. If the request passes all these checks, the smart contract is executed to generate a response that is sent back to the application. This response often contains the read and write sets. At this point, the transaction is not yet committed to the ledger unless a write transaction is executed.

3. *Inspection of proposal responses and assembly of endorsements:* The response

from the endorsing peers is inspected by the respective client application. If the transaction involved querying the ledger, the application displays the information. However, if the transaction involved writing, the client application inspects the endorsement policy before sending it to the *ordering service* to update the ledger. The client application then broadcasts the transaction proposal and response to the ordering service by including it in a *transaction message*. The ordering service then creates a block of transaction per channel.

4. *Transaction validation:* The blocks of transactions are then delivered to all the peers. As noted before, the transactions in the blocks are considered to be validated.

5. *Ledger Update:* Each peer applies the write sets of the transactions in the block thereby updating the ledger of each channel.

3.2.1.1 Endorsement Policies

An endorsement policy in Hyperledger Fabric refers to the peers that must agree on the results of a transaction before it can be added to the ledger. Hyperledger Fabric uses a small-domain language called `CLI` to define endorsement policies. Examples of endorsement policies include:

- All of the peers of each organization must endorse the transaction of type T, thus the endorsement policy T = (peer I) $\wedge$ (peer II) $\wedge$ ... (peer N).

- One or more peers of each organization can endorse the transaction of type Y, thus the endorsement policy Y = (peer I) $\vee$ (peer II) $\vee$ ... (peer N).

Endorsement policies are validation checks enshrined in the architecture of how the blockchain system handles transactions. The main advantage of this approach is that it allows developers to increase the privacy of data access stored on the blockchain.

3.2.1.2 Consensus Algorithm

Hyperledger Fabric employs a "*pluggable*" consensus algorithm; which means that the target use case application determines the algorithm to be used [8]. However, due to the inherent problems of distributed message passing such as nodes crashing and node collusion, validating peers in *v1.0* run a hybrid of the Practical Byzantine Fault Tolerance (PBFT) as the consensus algorithm to verify transactions [5]. Ideally, transactions are validated through the replicated execution of the smart contracts and the underlying assumption of Byzantine faults. It is assumed that among n peers, $f < n/3$ are faulty and may act arbitrarily to give false values but the rest execute the smart contract correctly [5]. The main advantage PBFT has over other distributed consensus algorithms such as PoW is that they have minimal latency and

can process numerous transactions per second, however, they do not scale easily in terms of nodes [8].

As any other blockchain system, the notion of consensus in Hyperledger Fabric refers to the agreement of the order of verified transactions written to the block. But most importantly, consensus also implies that a transaction has met all the checks that are imposed on it as the transaction completes the life cycle. As already stated, these checks include the check against endorsement policies set in the blockchain system and the check to prevent the double spending problem that might pose a threat to data integrity.

3.2.2 Membership Services and Identity Management

Hyperledger Fabric uses identities to manage network participants in the blockchain ecosystem, because of its private mode of operation. Ideally, these network participants include peers, client applications, and transaction orderers. Hyperledger Fabric supports a Membership Service Provider (MSP) that contains the security infrastructure for authentication and authorization of network participants.

Strictly speaking, identity management is achieved through the use of digital certificates. In particular, *enrollment* and *transaction authorization* is performed using Public Key Infrastructure (PKI) while *confidentiality* for smart contracts is achieved through homomorphic encryption. When connecting to the network, a peer obtains an *enrollment certificate* from the *enrollment certificate authority*. This certificate is used to authenticate the peer by the MSP. Upon authentication, a peer obtains a *transaction certificate* to enable it to issue transactions. The transaction certificate can be issued several times to the same peer allowing it to be pseudonymous. In addition, *channels* use symmetric encryption to provide confidentiality of the smart contracts and blockchain state.

3.3 Comparison

This section describes the key similarities and differences between the public blockchain implementation of Ethereum and the permissioned distributed ledger of Hyperledger project, called Fabric.

3.3.1 Similarities

Both Ethereum and Hyperledger Fabric share the same advantages that all of the blockchain solutions provide. Recording transactions through blockchain virtually eliminate errors caused by humans while at the same time protects the data from

prospective tampering. However, the fundamental value of a blockchain is that it enables a database to be directly shared without a central administrator. Both platforms use a consensus mechanism to ensure the nodes participating in each network stay synchronized. Therefore, neither Ethereum nor Hyperledger Fabric uses a central entity to verify the transaction taking place on their platforms. In addition, both systems support the full development and deployment of smart contracts. Thus, there is no need for third-party mediators or intermediaries for both platforms. Besides the time reduction in transactions, they both offer a reliable fault-tolerant network which is able to withstand malicious attacks, due to their decentralized nature.

3.3.2 Differences

The operational differences between Ethereum and Hyperledger Fabric stem from the target uses of the two flavors of blockchain systems [9]. Hyperledger Fabric was typically developed to be a modular blockchain that extends various use case applications. On the other hand, Ethereum seeks to provide a generic platform for all sorts of applications and modularity is not a key objective. Table 3.1 presents a summary of the key differences between Ethereum and Hyperledger Fabric.

Characteristics	**Ethereum**	**Hyperledger Fabric**
Description	Generic blockchain platform	Modular Distributed Ledger Technology (DLT)
Network operation	Public (permissionless)	Private (permissioned)
Type of transactions	Transparent	Confidential
Consensus algorithm	Mining based on PoW (Proof-of-Work)	Various Algorithms (Depending on target application)
Smart contracts	Solidity (`C++`, `Go`, `Python`)	Chaincode (`Go`, `Java`)
System currency	Ether	None
Mining reward	Yes	None

Table 3.1: Differences between Ethereum and Hyperledger Fabric.

3.3.2.1 Network operation

This is perhaps the most notable difference as it dictates the target use cases of the two blockchain systems. As already noted in Chapters 1 and 2, Hyperledger Fabric is a private or *permissioned* blockchain while Ethereum is a public or *permissionless* blockchain. This difference borders on how network participants are treated in the respective blockchain systems. Consequently, this difference plays a major role in how consensus is reached in these two blockchain implementations [9].

3.3.2.2 Consensus algorithm

In Ethereum, mining based on proof-of-work is used by all the peers to agree on the order in which transactions appear in the blockchain. As already noted, the main idea behind PoW as a distributed consensus algorithm is to limit the rate at which blocks are created by solving cryptographic puzzles. PoW suits the public nature of Ethereum due to the fact that any peer can participate in the mining process. If an adversary controls less than half of the total computing power present in the network, PoW prevents the adversary from creating new blocks faster than honest participants, thus inherently addressing the Sybil attack that is famous in anonymous networks [8]. However, PoW suffers from the performance of transaction processing when compared to other distributed consensus algorithms such as Byzantine Fault Tolerance (BFT) algorithms. This performance degradation arises from the need for all network participants to agree upon a common ledger and because the computational effort of PoW is both time and energy consuming [9]. In addition, ledger records are accessible by all network participants in Ethereum, which is problematic for use case applications that have strong privacy requirements.

In contrast, Hyperledger Fabric follows a novel approach to agree on the order in which transactions must appear on the blockchain. Typically, Hyperledger Fabric uses *execute-order-validate* scheme, which allows transactions to execute before the blockchain can reach an agreement on the order of the transactions. As already noted, Hyperledger Fabric uses a "*pluggable*" consensus algorithm but the most common is the hybrid Practial Byzantine Fault Tolerance (PBFT) algorithm. As opposed to Ethereum, nodes in the Hyperledger Fabric system assume different roles such as orderers and endorsement peers to help reach consensus. By separating the roles of nodes in this manner, Hyperledger Fabric is flexible enough to add an extra layer of permission by leveraging the blockchains identity management system thereby increasing the privacy of recorded data.

3.3.2.3 Smart contracts

Another key difference between Ethereum and Hyperledger Fabric is the use of smart contracts [4]. Ethereum smart contracts are written in a high-level contract-oriented language called Solidity. Smart contracts written in Solidity are compiled to bytecode that is executable on the EVM. Solidity allows developers to write distributed applications (dApps) that implement self-enforcing business logic inside smart contracts. Moreover, Solidity is statically typed, supports inheritance, libraries and complex user-defined types among other features. Ethereum supports also a Python-like language called Serpent, and a Lisp-like Language called LLL respectively.

Conversely, Hyperledger Fabric introduces another type of smart contracts, called "chaincode", which typically handles business logic agreed upon by members of the network, so it may be considered as a smart contract [5]. Every peer in the

Hyperledger Fabric network executes the chaincode, access the data stored in the distributed ledger and endorses new transactions. Hyperledger Fabric also offers an SDK for `Node.js`, `Java` and `Go`, mainly aiming in large integration projects. Chaincodes are written in `Go`, `Javascript` and `Java` and it is considered to be more flexible than a closed smart contract programming language, such as Solidity.

3.3.2.4 System currency

Section 3.1.2 describes how Ethereum uses *ether* as the built-in crypto-currency. It is used to pay nodes that take part in the consensus process as well as pay transaction fees whenever a smart contract is executed. Consequently, distributed applications that require monetary transactions can be built by taking advantage of this crypto-currency. On the other hand, Hyperledger Fabric does not require a built-in cryptocurrency as consensus is not achieved using the mining process. However, because chaincode is Turing complete, it is possible to create a native crypto-currency to operate in the Hyperledger Fabric ecosystem [9].

3.3.3 Conclusion

Ethereum and Hyperledger Fabric seek to solve the same problem i.e to provide a truly distributed way of recording transactions that enables trust in a trustless environment without the need for an intermediary. However, the two blockchain systems sit at different ends of the spectrum. Ethereum's public mode of operation allows it to be a strong candidate for transparent transactions but it sacrifices performance, scalability, and privacy. It is impossible to track or restrict the access to the blockchain, since the participants in public blockchains (such as Ethereum) are anonymous and they can join and leave the network whenever they want. Ethereum smart contracts make Ethereum blockchain extremely competent to the the business-to-consumer (B2C) market. On the other hand, Hyperledger Fabric uses its private mode of operation and modular architecture to solve privacy issues in the network system. Hence, it can provide solutions to sectors that require knowledge of who the members of the network are and who is accessing specific data. Furthermore, this modular architecture allows Hyperledger Fabric to have pluggable components such as the consensus algorithm to suit the requirements of the target application. Specifically, the commonly used BFT algorithm scales better than other consensus algorithms used in public blockchain and it allows the developer to increase the privacy of the application.

4

Implementation

In this chapter, we describe a Proof of Concept (PoC) implementation, consisting of a smart contract and distributed application (dApp), that allows a landlord and a tenant to make an agreement on a residential property. First, we define the basic rules and operations of the smart contract, according to the paper leasing contract. Thereafter, we describe the implementation of the smart contract and the dApp that handles the smart contract operations, using two different blockchain infrastructures, Ethereum and Hyperledger Fabric respectively.

4.1 Design and Specification

A paper contract is a legally binding arrangement between two or more parties that is enforceable by law as a binding legal agreement. Every contract must contain a detailed description (the terms of the contract) which specifies the obligations that parties have to each other. In order for the parties to agree on the terms of the contract, they have to sign it by giving their consent. Moreover, apart from the basic properties, a contract can have clauses, which can be either specific provisions or distinct articles within a contract. The role of contract clauses is to address a specific aspect related to the overall subject matter of the agreement. Contract clauses simply define the duties, rights, and privileges that each party has under the contract terms. Any contract concerning the law must go through several middlemen and third parties, thus paper contracts can take significant time and cost to become legally enforceable.

In order to implement a smart contract that corresponds to a paper contract, the terms of the contracts should be defined. The smart contract holds all the necessary information of the residential lease agreement and supports multiple operations that can be used by the two parties: landlord and tenant. In this PoC implementation, we assume that the person that deploys the smart contract becomes the landlord of the contract. However, we provide more fine-grained implementation details in Sections 4.2 and 4.3 respectively. Once the contract is deployed on the blockchain, its state is set to “Created”. In addition, the terms of the contract are clearly specified and proposed by the property owner.

4.1.1 System Description

The PoC implementation defines a contract between a landlord and a tenant. The contract stipulates that upon agreement, a tenant will have to pay an agreed amount on a monthly basis to the landlord as rent. Payments that are remunerated later than the agreed date attract a penalty fee as a percentage of the rent amount.

Our PoC implementation has two participants in the CRE business network, `Tenant` and `Landlord`, while `Houses` are the assets of this business network. Landlords own the houses that tenants seek to rent. We assume that tenants have searched for a house and the contract agreement marks the first entry transaction in the process flow. In addition, tenants and landlords are assumed to be authenticated as real players in the business network in the case of Hyperledger Fabric. The following are the functional requirements of the PoC implementation:

1. The landlord should create a contract that a tenant should review until an agreement is reached.
2. A tenant should be able to read the terms of the contract. Once agreed upon, only the parties purview to this agreement must retain access to the information generated as a result of the contract execution.
3. The tenant must remunerate the agreed rent amount. This transaction is modeled as a transfer of ether in Ethereum or transfer of an assumed account balance in Hyperledger Fabric.
4. The contract penalty should be adhered to once a late payment is received. The penalty fee is 5% of the principal rent amount unless otherwise negotiated. Furthermore, an event must be created notifying the landlord when a payment is missed by the tenant. Ethereum applies also extra fees (gas) for the smart contract execution, regardless of the extra fee for the penalty.
5. The tenant should be able to open negotiations on the terms of the contract during the validity of the contract and any new changes agreed to must be enforced on the blockchain.
6. Tenants must not be purview to contracts that do not concern them.

These requirements also encompass the terms of the contract. However, blockchain target requirements and assumptions will be exemplified in **Section 4.2** and **Section 4.3** respectively.

4.1.2 Design of the Smart Contract

Figure 4.1 illustrates the deployment of a smart contract by the landlord and the signing of the contract by the tenant. The smart contract can be in one of the four different states: `Created`, `Initiated`, `Terminated`, or `underReview`. After the creation of the smart contract, the state is initialized to the `Created` state and the tenant is able to read the contract terms.

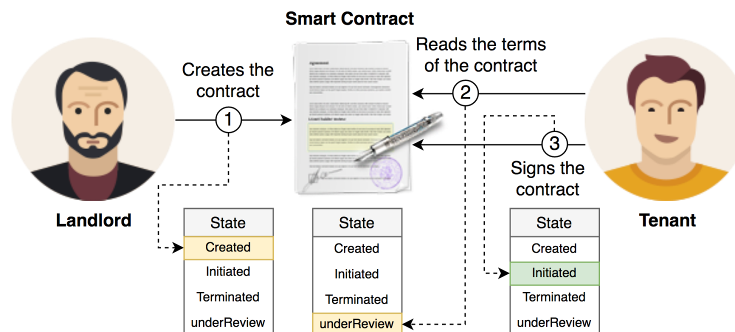


Figure 4.1: The deployment and the signing of the smart contract.

However, the tenant can propose new terms as a way of negotiating. The negotiating process moves the contract into the `underReview` state. This state puts the contract on hold until an agreement is reached between the negotiating parties. Similarly, after the agreement, the contract changes to the `Initiated` state and the lease operation is initiated. Consequently, the smart contract is now active and the parties involved are expected to follow the terms of the contract.

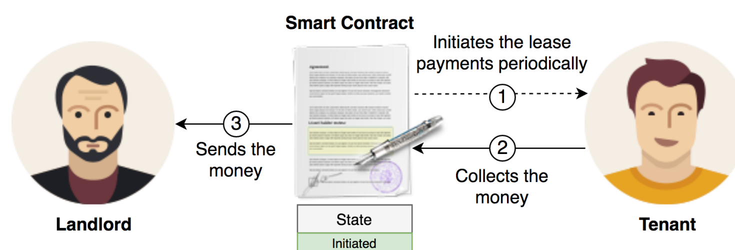


Figure 4.2: The payment process and the transfer of digital money.

Figure 4.2 illustrates the payment process initiated by the smart contract as the collection of the rent amount and transfer of the money from the tenant's account to the landlord's account. During this process, the two parties can use their blockchain addresses in order to transfer the money. In Ethereum, the built-in currency is used as a medium of exchange. On the other hand, participants in Hyperledger Fabric are modeled with an assumed account balance to achieve this functionality.

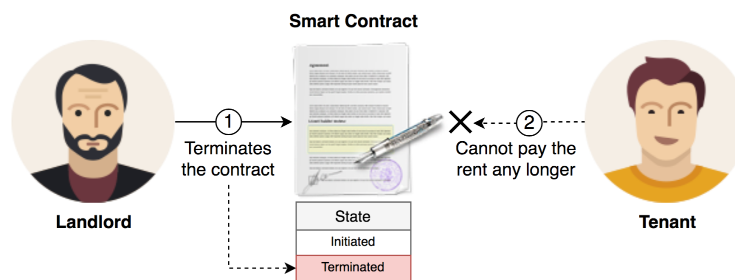


Figure 4.3: The termination of the agreement by the landlord.

Figure 4.3 illustrates the termination of the smart contract by the landlord. When the two parties have decided to end the agreement, or the tenant violated one of the terms, the landlord sets the smart contract status to `Terminated` and the contract is no longer active. The tenant is no longer able to interact with the smart contract or make any payments at all. After the smart contract is terminated it will continue to exist on the blockchain, but payments can not be made anymore, and the state will remain the same.

Role	Function	Description
Landlord	`readTerms()`	Read the description of the house and the terms of the contract
	`terminate()`	Terminate the lease of the contract
Tenant	`readTerms()`	Read the description of the house and the terms of the contract
	`signContract()`	Sign the contract and be obliged to the terms
	`payRent(month,year)`	Pay the rent corresponding to a specific month into the year

Table 4.1: Smart contract basic operations for landlord and tenant.

Table 4.1 lists the four basic leasing operations implemented in the smart contract. After the smart contract deployment, the implemented functions become available on the blockchain. In order to read the description of the house and the terms of the contract, someone can call the function `readTerms()`. Signing the contract is achieved by calling the `signContract()` function. The smart contract then is linked with his/her blockchain address and they are the only ones that can pay the rent using the function `payRent()`. The landlord can also terminate the smart contract using the function `terminate()`, and the tenant is no longer able to interact with the smart contract. For the purpose of this thesis we left out the case of the contract termination by the tenant, but it can added in the future work.

4.1.3 Design of the Negotiation Mechanism

Figure 4.4 illustrates the negotiation of the contract terms between the landlord and the tenant. The state of the smart contract should be `underReview`, in order for someone to propose any changes to the terms. First, one of the two participants makes a proposal and then the other one reads the proposed terms and either accepts or declines the changes. In this case, there are two potential scenarios: the tenant makes a proposal after reading the initial terms and then the landlord reviews the proposed terms or the opposite.

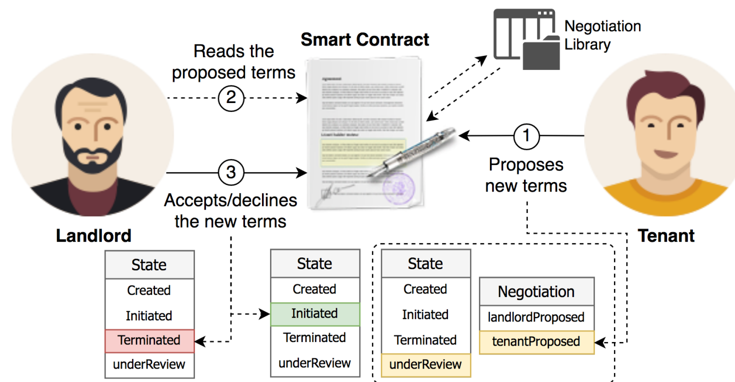


Figure 4.4: The smart contract negotiation process initiated by the tenant.

In order to distinguish these two scenarios, the smart contract also tracks the state of the negotiation process using two states: `tenantProposed` and `landlordProposed`. Once someone has accepted the new terms, the smart contract automatically adapts to the new terms and lease of the contract initiates. Thus, the state of the smart contract changes to `Initiated`. The negotiation library can be considered as a piece of code that can be called by other contracts, without the need to deploy it again.

Role	Function	Description
Landlord /Tenant	`proposeNewTerms()`	Propose one or a few changes in the terms of the contract
	`readNewTerms()`	Read the terms that have been proposed
	`declineNewTerms()`	Decline the proposed terms
	`acceptNewTerms()`	Accept the proposed terms

Table 4.2: Smart contract functions used for negotiations of the terms.

Table 4.2 lists the four functions implemented in the smart contract which are used for the negotiation process. Each of these function calls the appropriate function implemented in the negotiations library, thus any modifications the library does will be saved in the main smart contract's own storage. After the tenant has read the

initial terms, it is possible to make a proposal for one or more changes in the terms of the contract using the function `proposeNewTerms()`. The landlord then, can review the terms proposed, by the tenant by calling the function `readNewTerms()` and either accepts or declines the changes using the function `acceptNewTerms()` or `declineNewTerms()` respectively. If the landlord accepts the new terms the state of the contract changes to `Initiated`, while if they decline the proposed terms, the state changes to `Terminated`. The negotiation process can also be executed from the landlord's side, where they makes the proposal and the tenant can accept or decline the new terms.

4.1.4 Role-based Access Control Model

The related work of this thesis (section 2.5) includes two implementations for access control [27, 28] which are IoT-based systems operating on the blockchain technology. Additionally, Cruz *et al.* [29] proposed an RBAC system based on the Bitcoin network to provide an irrefutable proof of the role of the users. In order to implement a smart contact that restricts access to specific participants in the contract, we need to define an RBAC model that indicates the level of access each member has been given.

	State: `Created`			
Role	`readTerms()`	`signContract()`	`payRent()`	`terminate()`
Landlord	✓	✗	✗	✓
Tenant	✓	✗	✗	✗
Rest	✓	✗	✗	✗
	State: `Initiated`			
Role	`readTerms()`	`signContract()`	`payRent()`	`terminate()`
Landlord	✓	✗	✗	✓
Tenant	✓	✗	✓	✗
	State: `Terminated`			
Role	`readTerms()`	`signContract()`	`payRent()`	`terminate()`
Landlord	✓	✗	✗	✗
Tenant	✓	✗	✗	✗
	State: `underReview`			
Role	`readTerms()`	`signContract()`	`payRent()`	`terminate()`
Landlord	✓	✗	✗	✓
Tenant	✓	✓	✗	✗

Table 4.3: Access control list for the smart contract basic operations.

Table 4.3 shows the authorization of each party to execute the basic operations of the smart contract, while also includes the rest of the nodes on the network (state = `Created`) only for the case of the Ethereum blockchain (public). Each member has specific authorization to the functions implemented in the smart contract, according

to their role. The landlord and the tenant are able to read the terms at any time, but once the smart contract is initiated no other party is able to do so. To initiate the lease of the contract someone except the landlord has to sign the contract. The tenant, which is the party that has signed the contract, is the only one that is allowed to execute the payment of the rent.

	Negotiation: `landlordProposed`			
Role	`proposeNewTerms()`	`readNewTerms()`	`accept()`	`decline()`
Landlord	✗	✓	✗	✗
Tenant	✓	✓	✓	✓
	Negotiation: `tenantProposed`			
Role	`proposeNewTerms()`	`readNewTerms()`	`accept()`	`decline()`
Landlord	✓	✓	✓	✓
Tenant	✗	✓	✗	✗

Table 4.4: Access control list for the terms negotiation functions.

Table 4.4 describes the authorization of each party to execute the functions of the smart contract terms negotiation. After the deployment of the contract by the landlord the state of the negotiation process is initialized to `landlordProposed` and allows the tenant to make a request of one or more changes to the contract terms, which the landlord can accept or decline. Thus, both parties can initiate negotiations (two-way) but there is a restriction of who can accept or decline the proposal. Although, they both can read the new terms at any time.

4.2 Hyperledger Fabric Implementation

The implementation of the PoC makes use of `Hyperledger Composer` (from here on referred to as `Composer`), which is a rapid collaborative framework for developing applications that make use of Hyperledger Fabric. `Composer` is an open source project consisted of a modeling language called `Composer Modelling Language`, a user interface to quickly configure and test the business network called `Composer Playground` and `Command Line Interface (CLI)` tools to integrate `Composer` with an existing Hyperledger Fabric blockchain. The chaincode or Smart contracts are developed using `JavaScript` to house the transaction logic of the business network. Furthermore, the access control definition is written in a simplistic language. All these tools are bound together into a Business Network Definition (BND) archive and deployed on the Hyperledger Fabric blockchain.

4.2.1 System Setup

Figure 4.5 shows a simplified business model and the various resources of the CRE business network. The network is made up of *assets* and *participants*. In business terms, an *asset* is anything that is deemed to have value and can be exchanged in a business network. In the Hyperledger Fabric PoC implementation, the assets include the houses, contracts and contract terms that tenants and landlords have access to. The contract asset is the center of the business relation between the participating parties. It also relates the landlord, tenant, and contract to the terms that govern the relationship. A *participant* is a legitimate member of a business network. In the CRE network, participants include tenants that look to rent houses and landlords that own the property.

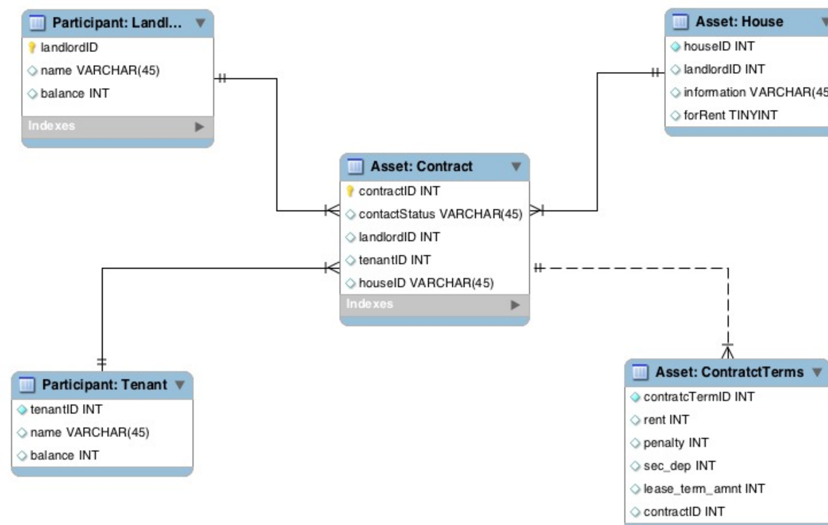


Figure 4.5: Simplified CRE business network model in Hyperledger.

The CRE network was modeled and tested using `Composer Playground` running in browser-only mode with no validating blockchain. The mock blockchain resides in the browser's history and allows the developer to test the transaction logic and access control list rules. Playground runs in a docker container and can either be fired from the loop-back address or from the IBM Bluemix server (similar to the Remix Solidity IDE). The modeled business network was later deployed to a validating blockchain on a development environment consisting of two peers that handle transactions arising from the two sets of participants. The endorsement policy is set in such a way that both peers sign transactions before they are executed and subsequently written to the peers' permanent storage. Similarly, the `Composer CLI` tools install the chaincode and a supporting SDK (in this case, the NPM JavaScript runtime environment) in order to issue and execute transactions.

4.2.2 Transaction Processing

Transactions allow participants to interact with the assets. Chaincode consisting of logic from the various operations of the CRE application are first modeled using the `Composer Modeling Language` and later implemented as functions using JavaScript. The modeled transactions are implemented with the help of transaction processor functions. The transaction processor function declaration contains metadata and a qualified JavaScript function name used to call the function. The metadata contains a parameter declaration used to call the function and the namespace resource of the transaction in the model file. Finally, the declaration contains a `@transaction` tag that identifies the subsequent code as a transaction processor file. Listing 4.1 describes how a function to handle the functionality of paying the rent is defined in a chaincode file.

```
/**
 * transaction to pay rent by the tenant
 * @param {estate.payRent} cont
 * @transaction
 */
 async function payRent(cont) {
    const amount = cont.amount;
    const cont_rent = cont.cont_terms.rent;
    const ten_balance = cont.tenant.balance;

    if(amount < cont_rent || amount < ten_balance){
        console.log("Insuficient balance")
    }else{
        //apply the rent
        cont.landlord.balance += cont.amount;
        cont.tenant.balance -= cont.amount;

        //Update the tenants balance
        const tenant = await await
        getParticipantRegistry('estate.Tenant');
        await tenant.update(cont.tenant);

        //update the landlords balance
        const landlord = await await
        getParticipantRegistry('estate.Landlord');
        await landlord.update(cont.landlord);

        console.log("Rent of: "+amount+" has been paid");
    }
 }
```

Listing 4.1: Sample transaction processor function to make payments.

4.2.2.1 Access Restriction

Hyperledger Fabric offers a fined grained security mechanism to protect the resources of the business network. There are two levels of security in this implementation:

- *Hyperledger Fabric administrator:* This is the administrator of the blockchain and was defined when installing the Hyperledger Fabric on the development computer. The blockchain administrator oversees the entire network and can add/delete peers as well as issue/revoke security credentials of participating organizations. The blockchain network has a peer node for each organization named `peer0.org.tenant.com` and `peer1.org1.landlord.com`. Furthermore, the network consists of a single certification authority (CA) running at port 7054 and a single ordering service running at port 7050. Finally, there is one channel created in the blockchain system and both peers joined it.

- *Business network administrator:* This administrator is created when the business network administrator is deployed to the blockchain, which is responsible for authentication of participants of the network.

As already alluded to, identity management in Hyperledger Fabric is very cardinal. In this implementation, ID cards are issued to all the participants by the business network administrator. An ID is a collection of all the necessary authentication information that allows a participant to connect to a business network. As an example `tenant1@estate-network` is a card issued for participant `Tenant1`.

```
rule R1b_ReadTerms {
  description: "Tenant can read terms only if the ID is
  equivalent to participant"
  participant(t): "estate.Tenant"
  operation: ALL
  resource(c): "estate.readTerms"
  condition: (c.owner.getIdentifier() == t.getIdentifier())
  action: ALLOW
}
```

Listing 4.2: Access Control List (ACL) rule to restrict access to the read terms function.

Following the role-based rules defined in Section 4.1.4, necessary rules have been included in the BND file. Listing 4.2 shows one of the access control list rules used in the implementation. In this listing, the participant clause relates to the `Tenant` participant and operation clause suggests that if the rule is met, then all the CRUD (Create, Read, Update, Delete) operations are allowed. Furthermore, the resource being restricted in this case being the `readTerms` function is specified under the resource clause on the condition that the ID contained in the contract is equivalent to the ID attempting to execute the `readTerms` function. There are four rules in

total and the reader is invited to refer to the Appendix for more details about the source code.

4.2.3 Negotiation Mechanism

In our PoC implementation, we implemented four additional functions in the smart contract, one for each of the operations of the negotiation mechanism, and two extra functions in the Ethereum library implementation that can be called by the smart contract. Unlike the library implementation in Ethereum, Hyperledger Fabric implements the negotiation scheme as transactions modeled and implemented in the smart contract, thus only the basic four functions are implemented. This scheme is more secure as extra access controls can be applied to the modeled transactions and fulfills the bilateral negotiation between the tenant and landlord. A more detailed description of the negotiation logic flow can be found in **Section 4.3.3.2**, but in the case of Hyperledger Fabric the flow is made up of four transactions that move the contract between the `Initiated` or `Terminated` states. The negotiation scheme is achieved using the following functions:

- `readTerms():` The function retains the terms of the house set by the landlord. The tenant and/or landlord can call this function once a contract has been created and is in the `underReview` state. The access control rule only allows the assigned landlord and tenant to call this function. Consequently, the tenant or landlord can choose to either accept, decline or propose new terms.

- `proposeNewTerms():` This function allows both the tenant and landlord that are assigned to a contract to propose new terms. The contract state is changed to the `underReview` state. Similarly, the appropriate control rule allows only the assigned parties to invoke this function. Furthermore, once terms are proposed, an event is created to notify the other party.

- `acceptNewTerms():` The function is invoked when the proposed terms of the contract are acceptable to either parties. The state of the contract is moved to the `Initiated` state.

- `declineNewTerms():` The function is invoked when the proposed terms of the contract are not acceptable to either parties. The state of the contract is moved to the `Terminated` state.

4.3 Ethereum Implementation

In this section, we describe the implementation of our PoC for residential property agreements in Ethereum platform. First, we present the developing tools and thereafter we continue with the description of the implementation in Solidity.

4.3.1 System setup

In order to develop and deploy smart contracts in the Ethereum blockchain, an appropriate setup is required to allow the interaction between the developer's computer machine and the rest of the devices connected to the blockchain network. The first step is to set up an Ethereum node and connect it to the Ethereum network.

4.3.1.1 Ethereum Node

An Ethereum node, in its simplest form, can be viewed as any device that is running the Ethereum protocol [4]. Ethereum nodes are typically running on any type of computers, such as desktops or laptops. Any computer that is running the Ethereum protocol, is actually connected to the Ethereum blockchain network and is running a node. Ethereum nodes are also connected to other nodes in the network, and they have direct access to the blockchain, and can contribute to the mining process, send transactions, and deploy smart contracts.

4.3.1.2 Remix Web Browser IDE

Remix is a browser-based compiler and IDE that enables developers to write Ethereum smart contracts with Solidity language and to debug transactions. Typically it is used for medium to small sized smart contracts and provides plenty of useful features to the developers, such as:

- Step by step integrated debugger for monitoring the instructions, variables, and calls to data or the stack.
- Generating warnings for unsafe code, gas cost, whether functions could be constant, and more.
- Integrated testing and deployment environment.
- Static analysis, syntax, and error automatic highlighting.
- Support for injected Web3 objects (local or remote Ethereum node).

Remix is an all-in-one web-based solution for developing and deploying Ethereum smart contracts. Remix provides all the necessary infrastructure to write, debug and test the smart contract making the development process effortless. Thus, developers can test their source code without setting up Ethereum nodes or wallets to initiate the sample transactions.

4.3.2 Building the Smart Contract

The Solidity code is statically typed, supports inheritance, libraries and complex user-defined types among other features. Solidity is a contract-oriented, high-level language that is designed to target the Ethereum Virtual Machine (EVM). At the time of writing this thesis, the latest Solidity version was `0.4.21`. Solidity source files are annotated with a so-called version `pragma` to reject being compiled with future compiler versions that might introduce incompatible changes.

4.3.2.1 State Variables

Before implementing any functions in the smart contract, we defined the state variables which are permanently stored in contract storage, once the contract is deployed. Our PoC implementation consists of two participants, the landlord and the tenant, so we keep their blockchain addresses in two `address` variables, namely `landlord` and `tenant`. Each smart contract handles one house, so we also defined several `string` variables to keep the house details and `unsigned integer` variables for the leasing terms respectively. The house details include the address, the zip code and the city of the house, while the leasing terms consist of the contract deployment date, the amount of rent, the duration, the last day of the month that the tenant can pay the rent and the penalty fee for the delayed payments. Moreover, the states of the contract and the negotiation process are also defined as `enum` types which is a user-defined type. In order to keep track of the payments we defined a data structure `RentPaid` with the following attributes: `paymentID`, `paymentDate`, `rent`, `month` and `year`. In addition, we defined a list of this struct to keep a history log of all payments.

4.3.2.2 Functions

The landlord deploys the smart contract using the contractor of the smart contract, which is a public function called `HouseLeasing()`. This function has the same name as the smart contract and it is called along with the house details and the leasing terms as input arguments. Public functions can be accessed by all nodes, while private functions are only accessible by the contract itself. The state variables are modified accordingly and the two types of states are initialized to the default values (state = `Created` and negotiation = `lanlordProposed`). The leasing operation supports four operations, thus we implemented four public functions accordingly:

- `readTerms()`: Returns the house details and the initial terms that the landlord has set. This function is accessible by everyone once the contract is created, but if a potential tenant makes a call to the function, the smart contract set on hold (state = `underReview`) until they finish their procedure (read/negotiate/sign). After that, the rest of the parties are excluded from the smart contract and

only the landlord and the potential tenant can use the function. In order to restrict the access in this function, we implemented appropriate modifiers that check whether the blockchain address of the party that makes the call is the landlord or the tenant.

- `signContract()`: Sets the agreement between the landlord and the tenant and binds the tenant to the terms. Once the potential tenant has read the terms of the smart contract using the function `readTerms()`, his/her blockchain address has been stored in the smart contract and they are the only one that have access to the function `signContract()`. No other blockchain address can call this function (using modifiers), and once it is called, it initiates the lease of the contract (state = `Initiated`). After this point it is no longer accessible, preventing the overwriting of the tenant's blockchain address.

- `payRent(month,year)`: Takes as input the month and the year of the payment initiated by the tenant and transfers the amount of digital money (ether) from the tenant's blockchain address to the landlord's one. Besides the input arguments, the definition of the function contains also the keyword `payable`, which is a modifier allowing the function to receive ether. Once the contract is initiated, the function allows only the tenant to use it in order to make payments. Moreover, an extra modifier checks whether the amount of money that is about to be transferred corresponds to the amount of rent that is agreed by the two parties. The date of the payment is also checked by the function and if the due day was earlier it checks if the amount transferred includes also the penalty fee for delayed payments. The amount of ether that is transferred (in wei) is actually the value of the message.

- `terminate()`: Terminates the lease of the contract and sets the state of the contract to `Terminated`. The landlord can use this function if the tenant violated any of the rules, otherwise, it is called automatically after the period of time the two parties have agreed upon. If there is an unpaid rent, this function will be called once the tenant completes the transfer of money.

4.3.2.3 Events

Solidity also supports events, allowing usage of the Ethereum Virtual Machine logging facilities. Specifically, they can call JavaScript callbacks in the user interface of a dApp, which listens to these events. When they are called, they cause the arguments to be stored in the transaction's log (a special data structure in the blockchain), thus they are inheritable members of contracts. Events are emitted using the keyword `emit` followed by the name of the event and the arguments. Any such invocation can be detected from the JavaScript API by filtering for `Deposit`. In our implementation, we defined four events, one for each function, as follows: `termsRead()`, `contractSigned()`, `rentPaid()`, and `contractTerminated()`.

4.3.2.4 Access Restriction (Modifiers)

Restricting access is a common pattern for smart contracts, but it is impossible to restrict any human or computer from reading the content of the smart contract transactions. Although, it is possible to restrict who can make modifications to the state of the smart contract or make calls to its functions. In particular, modifiers can be used by functions to allow only certain participants to execute the function, thus this is a way of checking a condition prior to executing the function. Listing 4.3 shows how function modifiers are defined and used in the Solidity programming language.

```
/* Smart contract constructor */
function Example() {
   /* State variables initialization */
   creator = msg.sender;
}

/* Modifier definition */
modifier onlyCreator() {
    /* If the condition is not met then throw an exception */
    if (msg.sender != creator) throw;
    /* or else just continue executing the function */
    _;
}

/* Adding modifier to the function */
function kill() onlyCreator {
    selfdestruct(creator);
}
```

Listing 4.3: Example modifier used to check a condition prior to executing the function.

In our PoC implementation, we implemented five function modifiers that they are used by our basic functions, in order to restrict access to them:

- `_require(bool condition)`: A modified version of the pre-defined 'guard' function `require(condition,'error')`, which takes as an input one Boolean condition, and returns an error if the condition does not hold. The `require()` pre-defined function takes two inputs, the condition and the error message, while our modified function uses only the condition as an input and returns the same result as the pre-defined function.

- `onlyLandlord()`: Checks if the Ethereum address that tries to call the function, corresponds to the registered landlord's Ethereum address. If not it returns an error message.

- `onlyTenant()`: Checks if the Ethereum address that tries to call the function, corresponds to the registered tenant's Ethereum address. If not it returns an error message.

- `isState(State state)`: Compares the current state of the smart contract with a given input of the contract state (`Created`, `Initiated`, `Terminated`, or `underReview`). If the contract is not in the same state as the given input, it returns an error.

- `isNegotiationState(NegotiationState negotiationState)`: Compares the current state of the smart contract negotiation process with a given input of the negotiations state (`landlordProposed` or `tenantProposed`). If the negotiations are not in the same state as the given input, it returns an error.

4.3.2.5 Helper Functions

In order to clarify and evaluate the leasing operations of our smart contract, it is essential to use appropriate functions that return basic values. Therefore, we implemented several helper functions to read the values from the blockchain at any time, in a similar way setter and getter methods are implemented in `Java`:

- `getHouseDetails()`: Returns the details of the house property, such as the address, the zip code and and the city.

- `getLandlord()`: Returns the landlord's Ethereum address.

- `getTenant()`: Returns the tenant's Ethereum address.

- `getContractAddress()`: Returns the smart contract's Ethereum address as a string.

- `getContractCreated()`: Returns the timestamp of the smart contract's creation, or more specifically the timestamp of the smart contract's first block.

- `getState()`: Returns the state of the smart contract.

- `getNegotiationState()`: Returns the state of the negotiation process.

- `getBalance()`: Returns the available balance (in `wei`) of the Ethereum address that makes the call to the function.

- `getRentPaid()`: Returns the history log of the rent payments made by the tenant.

4.3.3 Negotiation Mechanism

In this section, we describe the implementation of the mechanism for negotiation of the smart contract terms. The negotiation process is bilateral, thus both sides can propose and review the terms of the contract.

4.3.3.1 Libraries in Solidity

In Solidity, libraries have no storage and do not hold any assets (*ether*). A Solidity library can be considered as a piece of code that can be called from any smart contract written in Solidity, without the need to deploy it again. This concept allows developers to save substantial amounts of gas while the blockchain is not contaminated with repetitive code. The fact that different smart contracts can rely on the same library that has been already deployed, creates a more clean and secure environment across the Ethereum blockchain. Solidity libraries can contain logic and are used to extract code away from the smart contracts for maintainability and reuse purposes. Nonetheless, libraries are separate storage contracts, thus, an extra call to the storage contract is necessary besides the one from the action contract through the interface. Moreover, the developers should always check a public library before using it in their smart contract, in order to ensure that does not pose any security risks.

4.3.3.2 Library Implementation

Libraries are defined using the keyword `library`, instead of `contract` that is used in the smart contract. Thereafter, the library is imported in the smart contract and the smart contract can call the functions implemented in the library. In our PoC implementation, we implemented four additional functions in the smart contract, one for each of the operations of the negotiation mechanism, and two functions in the library that can be called by the smart contract:

- `proposeTerms(rent,duration,day)`: Takes as input the proposed rent, duration and last day to pay and initiates a proposal using the library's function `makeProposal(rent,duration,day)`. This function can only be used if the smart contract state is `underReview`, while the initial terms of the contract are not altered, and the proposed terms are kept in a different structure within the library. Similarly to the four basic functions, we used modifiers to check whether the blockchain address of the party that makes the call is the landlord or the tenant.

- `readNewTerms()`: When either the landlord or the tenant has proposed new terms, the other party can use this function to read the proposed terms, using the library's function `readProposal()`. If one or more terms have been proposed, the library function returns the proposed terms and the smart contract

function shows them to the user respectively.

- `acceptNewTerms()`: If one or more terms have been proposed, the party that did not make the proposal can accept the requested terms. The state of the smart contract is set to `Initiated`, and the negotiation process is terminated.

- `declineNewTerms()`: If one or more terms have been proposed, the party that did not make the proposal can decline the requested terms. The state of the smart contract is set to `Terminated`, and both the smart contract and the negotiation process are terminated.

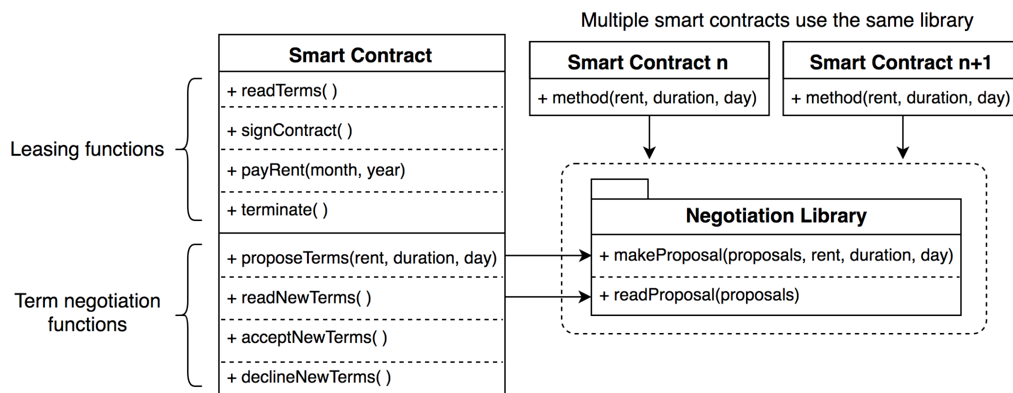


Figure 4.6: Interaction between the smart contract and the library.

Figure 4.6 shows the communication between the smart contract and the negotiation library by listing the methods defined. The library uses a data structure to keep track of the proposals, by storing the rent, duration, last day to pay and the date that the proposal had been made. Moreover, we defined two functions to handle the proposals:

- `makeProposal(proposals,rent,duration,day)`: Takes as input the proposed rent, duration and the last day to pay, and stores the information along with the timestamp of the proposal in the proposal data structure. It categorizes the proposals by different smart contracts, ensuring that proposal from different smart contracts and parties are not mixed together.

- `readProposal(proposals)`: Returns the contents of the proposed terms that are stored in the proposals. Similar to the previous function, participants from other smart contracts cannot read the proposals of each other negotiation process.

4.3.4 Wallet Recovery

Ethereum's public mode of operation allows parties to freely join and leave from the blockchain, lacking a verification mechanism to its users. Thus, participants can

come and go whenever they want without the need of subscription. Therefore, if a participant in a smart contract leaves the blockchain, and wants to regain full access again using a new identity, the blockchain is not aware if the person trying to join is the same person or another. In our PoC implementation, the smart contract is always linked with two Ethereum addresses, the tenant's and the landlord's. If one of those two loses access to their Ethereum address, they instantly lose access to the smart contract as well. Losing access to the Ethereum address simply means that the participant lost access to the wallet that handles the specific Ethereum address or the Ethereum address of the wallet has changed. Thus, we designed and implemented an additional mechanism to identify if a person trying to restore access is the same person who left the blockchain network before.

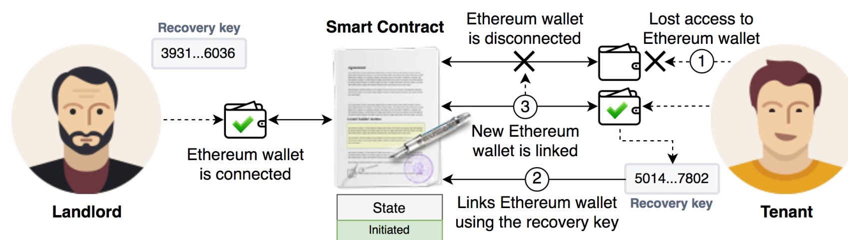


Figure 4.7: Tenant links a new wallet using the private recovery key.

Figure 4.7 illustrates an example of linking a new wallet (Ethereum address) to the smart contract by the tenant. In our implementation, we assumed that one Ethereum wallet handles one Ethereum address, and not multiple addresses. Some wallets are able to generate different Ethereum addresses for every transaction for security reasons, but this does not apply to our case. A recovery key is automatically generated and sent to the landlord after the smart contract becomes available on the blockchain (state = `Created`). In particular, the computation of the key (256-bit number) is the Ethereum-SHA-3 (Keccak-256) hash of the current block (the block of the smart contract deployment). The recovery key for the tenant is generated and becomes available only to this party, once the terms have been read and the contract state is set to `Initiated` or `underReview`. The two parties store their recovery keys in a safe place (e.g. paper - not digital storage), and they can use them at anytime to revoke their access to the smart contract. The wallet recovery mechanism has two functions one for each of the participants:

- `recoverLandlord(key)`: Takes as an input the landlord's recovery key and modifies the landlord's Ethereum address associated with the smart contract.
- `recoverTenant(key)`: Takes as an input the tenant's recovery key and modifies the tenant's Ethereum address associated with the smart contract.

5

Evaluation

In this chapter, we describe the evaluation process of the Proof of Concept (PoC) proposed in chapter 4, which includes a Role-based Access Control (RBAC) mechanism and a novel library implementation for negotiation of the smart contract terms. First, we describe the fulfilment of our evaluation methodology and second we present the findings according to this criteria. Further, we perform a performance evaluation on Ethereum implementation investigating the gas consumption in our PoC implementation.

5.1 Evaluation Criteria

In this section we describe the evaluation criteria according to the methodology defined in Chapter 1. Table 5.1 lists the evaluation criteria for Ethereum and Hyperledger Fabric. They are divided into three categories according to the target being evaluated. Firstly, we evaluate the leasing operation of both implementations by running multiple scenarios for all of the leasing functions. These scenarios include also the evaluation of the successful deployment of the smart contracts in the two different target blockchain networks. Secondly, we evaluate the functionality of the RBAC mechanism with dedicated scenarios that exploit the access to the smart contract contents. In such scenarios, we expect to find how the two PoC implementations handle participants that have no authorization to access the contents. Lastly, we evaluate the negotiation library by executing different scenarios from both sides (landlord and tenant), where requests are sent bilateral.

Explicitly in Ethereum, we also evaluate the functionality of the wallet recovery mechanism for participants that lost their access to the wallet that is linked to the smart contract. No such mechanism is necessary in the Hyperledger Fabric implementation, because the business network administrator handles all of the subscriptions of the participants in the network.

Blockchain	Target	Scenario Description
Ethereum and Hyperledger Fabric	Leasing operation	Evaluate the functionality of the leasing utility. Check the basic functions for reading the terms, signing and terminate the contract, and paying the rent.
Ethereum and Hyperledger Fabric	RBAC mechanism	Evaluate the access restriction to certain parties. Check if non-authorized participants are denied access to the smart contract functions and contents.
Ethereum and Hyperledger Fabric	Negotiation library	Evaluate the functionality of the automated process for terms negotiation. Check if the negotiation library can handle the proposals of both parties.
Ethereum	Wallet recovery	Evaluate whether a participant can gain access to the smart contract using a new Ethereum wallet.

Table 5.1: Evaluation criteria for Ethereum and Hyperledger Fabric.

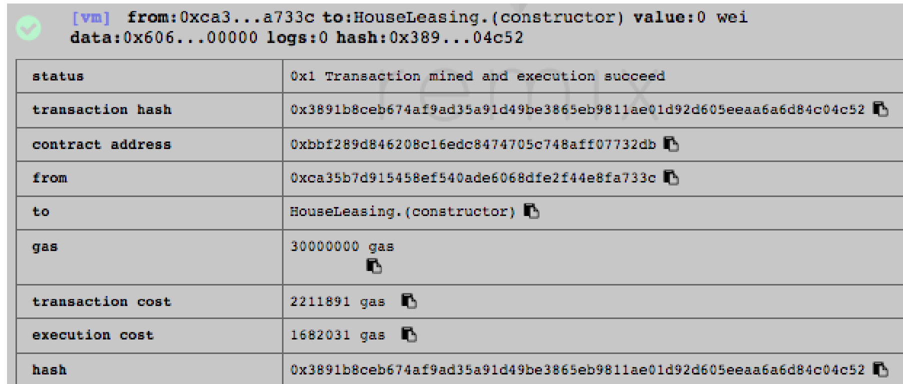
5.2 Descriptive Evaluation

Our main evaluation objective is the descriptive evaluation of our PoC implementation, performed using the functional criteria. We evaluate the two PoC applications implemented in two different blockchain systems and then we present our findings. For more details on the source code, the reader is directed to the appendices, where the source code of both implementations (Ethereum and Hyperledger Fabric) is available.

5.2.1 Leasing Operation

The evaluation of the smart contract leasing operation was held using scenarios of two participants, the landlord and the tenant. The most fundamental action is the deployment of the smart contract, performed by the landlord. Once it has been deployed, the potential tenant can read, negotiate, sign and pay the rent of the contract. We conducted multiple scenarios where one participant (landlord) initiates the contract lease and then the two participants try to call all of the functions available by the contract.

Figure 5.1 shows the smart contract deployment on Ethereum blockchain using the Remix IDE. In this particular scenario the landlord owns an apartment located in Gibraltargatan 36, Gothenburg 41258, and uses the Ethereum address `0xca35b7d915458ef540ade6068dfe2f44e8fa733c`.

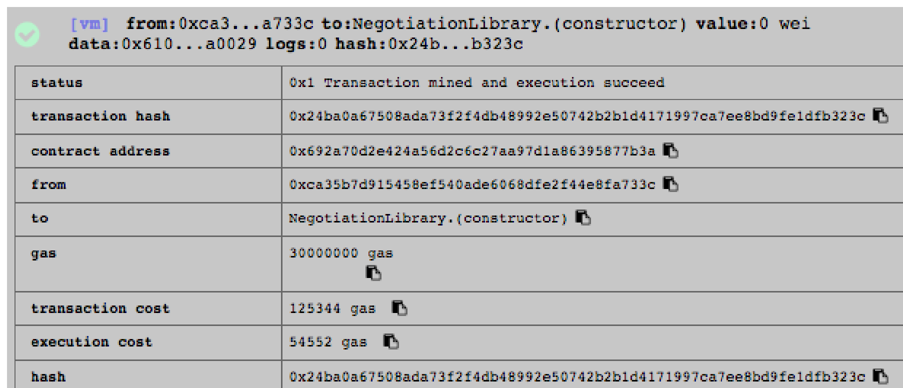


[vm] from:0xca3...a733c to:HouseLeasing.(constructor) value:0 wei
data:0x606...00000 logs:0 hash:0x389...04c52

status	0x1 Transaction mined and execution succeed
transaction hash	0x3891b8ceb674af9ad35a91d49be3865eb9811ae01d92d605eeaa6a6d84c04c52
contract address	0xbbf289d846208c16edc8474705c748aff07732db
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	HouseLeasing.(constructor)
gas	30000000 gas
transaction cost	2211891 gas
execution cost	1682031 gas
hash	0x3891b8ceb674af9ad35a91d49be3865eb9811ae01d92d605eeaa6a6d84c04c52

Figure 5.1: Smart contract deployment on Ethereum blokchain.

The smart contract is linked with the negotiation library, so therefore the library is automatically deployed after the smart contract creation. Figure 5.2 shows the negotiation library deployment on Ethereum blockchain using the Remix IDE. Remix IDE offers multiple Ethereum addresses to test transactions on the blockchain. In our case, we set the gas limit to 30 `Mwei` (=`30000000 wei`) and then we initiated the transaction. The transaction was verified by the network (miners) and the smart contract has been deployed successfully on the blockchain with a permanent Ethereum address.



[vm] from:0xca3...a733c to:NegotiationLibrary.(constructor) value:0 wei
data:0x610...a0029 logs:0 hash:0x24b...b323c

status	0x1 Transaction mined and execution succeed
transaction hash	0x24ba0a67508ada73f2f4db48992e50742b2b1d4171997ca7ee8bd9fe1dfb323c
contract address	0x692a70d2e424a56d2c6c27aa97d1a86395877b3a
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	NegotiationLibrary.(constructor)
gas	30000000 gas
transaction cost	125344 gas
execution cost	54552 gas
hash	0x24ba0a67508ada73f2f4db48992e50742b2b1d4171997ca7ee8bd9fe1dfb323c

Figure 5.2: Negotiation library deployment on Ethereum blokchain.

In contrast, the deployment of the contract and creation of the participants (tenants and landlords) in Hyperledger Fabric is the sole responsibility of the business network administrator as already alluded to in Section 4.3.1. Figure 5.3 illustrates the administrator deploying the business network to an existing Hyperledger Fabric

blockchain using Composer. The business network bundles the model and smart contract files into a business definition archive.

```
Malamas-MacBook-Air:estate malamakasanda$ composer archive create -t dir -n .
Creating Business Network Archive


Looking for package.json of Business Network Definition
        Input directory: /Users/malamakasanda/HyperledgerComposer/estate

Found:
        Description: Real Estate Application
        Name: estate
        Identifier: estate@0.0.1

Written Business Network Definition Archive file to
        Output file: estate@0.0.1.bna

Command succeeded

Malamas-MacBook-Air:estate malamakasanda$ composer runtime install --card PeerAdmin@hlfv1 --businessNetworkName estate
✔ Installing runtime for business network estate. This may take a minute...

Command succeeded
```

Figure 5.3: Deployment of the business network to Hyperledger Fabric.

Using the Composer Playground (similar to the Remix IDE for Ethereum), the business network administrator is able to call the `createTenant` and `createLandlord` functions to instantiate the necessary participants. Equally, Figure 5.4 shows the creation of two network participants on the business network along with their identifications, names, and balances.

Submit Transaction

Transaction Type: createLandlord

JSON Data Preview

```
1  {
2    "$class": "estate.createLandlord",
3    "landlordID": "L1",
4    "name": "John Doe",
5    "balance": 100
6  }
```

☐ Optional Properties

Transaction Type: createTenant

JSON Data Preview

```
1  {
2    "$class": "estate.createTenant",
3    "tenantID": "T1",
4    "name": "Craig",
5    "balance": 1000,
6    "forRent": false
7  }
```

☐ Optional Properties

Just need quick test data? Generate Random Data | Cancel | Submit

Figure 5.4: Creation of two network participants on Hyperledger Fabric.

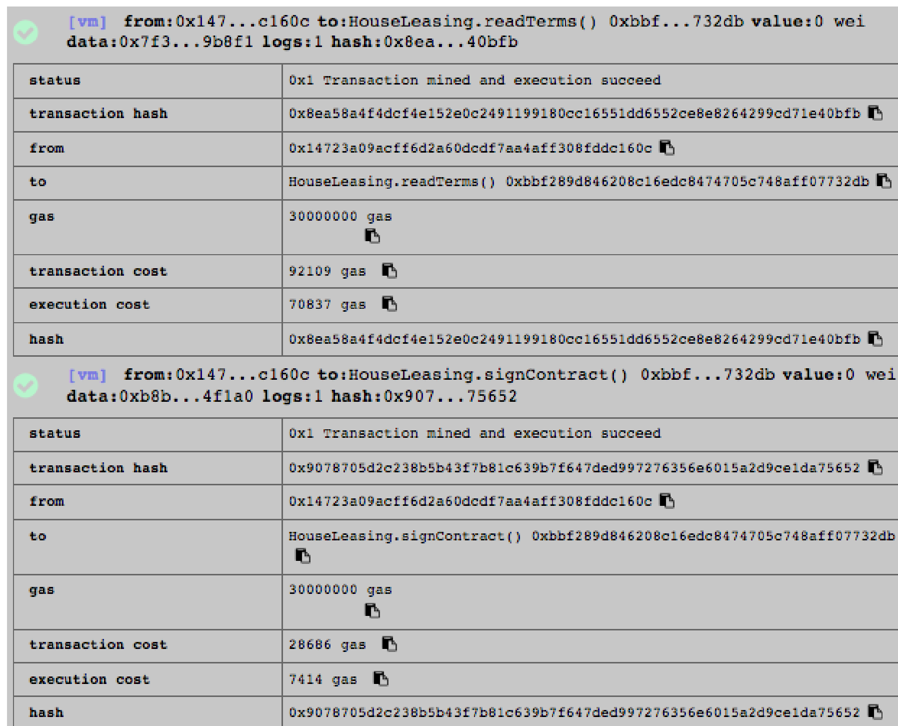
5.2.2 RBAC Mechanism

The evaluation of the RBAC mechanism was done by executing illegal transactions where non-authorized participants try to execute functions in the smart contract. In this case, appropriate function modifiers have been used to block any illegal transaction. Besides the restriction to non-authorized participants in the smart contract, the function modifiers protect also the active members of the smart contract. They are allowed to execute only certain functions, thus their roles are immutable during the lease. Table 5.2 lists the basic functions of the smart contract leasing operation and the corresponding function modifier (Ethereum) or rule (Hyperledger Fabric) used for access restriction.

Function	**Blockchain**	**Motivation**
`readTerms()`	Ethereum	The function modifier `_require(condition)` restricts the access to this function. The condition allows only the landlord and the tenant.
	Hyperledger Fabric	The rule `R1b_ReadTerms{}` restricts the access to this function (only landlord and tenant).
`signContract()`	Ethereum	The function modifier `onlyTenant()` restricts the access to this function. It allows only the tenant.
	Hyperledger Fabric	The rule `R2b_SignContract{}` restricts the access to this function (only allowing the interested tenant).
`payRent(month,year)`	Ethereum	The function modifier `onlyTenant()` restricts the access to this function. It allows only the tenant.
	Hyperledger Fabric	The rule `R3b_PayRent{}` restricts the access to this function (only tenant).
`terminate()`	Ethereum	The function modifier `onlyLandlord()` restricts the access to this function. It allows only the landlord.
	Hyperledger Fabric	The rule `R4b_Terminate{}` restricts the access to this function (only landlord).

Table 5.2: Evaluation of the RBAC mechanism for the leasing operation.

An Ethereum case scenario in Section 5.2.1.1 shows the deployment of the smart contract (and the library) by the landlord (0xca3...a733c). A potential tenant (0x147...c160c) tries to read the terms and then sign the contract. Figure 5.5 shows the two transactions initiated by the potential tenant. The potential tenant's Ethereum address is stored safely in the smart contract once they open the terms.


[vm] from:0x147...c160c to:HouseLeasing.readTerms() 0xbbf...732db value:0 wei data:0x7f3...9b8f1 logs:1 hash:0x8ea...40bfb

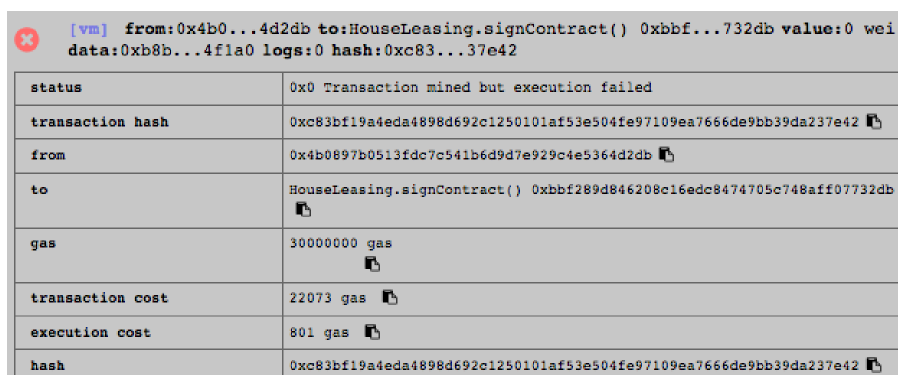
status	0x1 Transaction mined and execution succeed
transaction hash	0x8ea58a4f4dcf4e152e0c2491199180cc16551dd6552ce8e8264299cd71e40bfb
from	0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
to	HouseLeasing.readTerms() 0xbbf289d846208c16edc8474705c748aff07732db
gas	30000000 gas
transaction cost	92109 gas
execution cost	70837 gas
hash	0x8ea58a4f4dcf4e152e0c2491199180cc16551dd6552ce8e8264299cd71e40bfb

[vm] from:0x147...c160c to:HouseLeasing.signContract() 0xbbf...732db value:0 wei data:0xb8b...4f1a0 logs:1 hash:0x907...75652

status	0x1 Transaction mined and execution succeed
transaction hash	0x9078705d2c238b5b43f7b81c639b7f647ded997276356e6015a2d9ce1da75652
from	0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
to	HouseLeasing.signContract() 0xbbf289d846208c16edc8474705c748aff07732db
gas	30000000 gas
transaction cost	28686 gas
execution cost	7414 gas
hash	0x9078705d2c238b5b43f7b81c639b7f647ded997276356e6015a2d9ce1da75652

Figure 5.5: The potential tenant reads the terms and signs the contract.

If another potential tenant using a different Ethereum address tries to sign the contract without first reading the terms (or if someone else has already opened the terms), the first will not be able to deploy the transaction on the blockchain. Figure 5.6 shows an illegal transaction from a potential tenant (0x4b0...4d2bd) that has no authorization to sign the contract.


[vm] from:0x4b0...4d2db to:HouseLeasing.signContract() 0xbbf...732db value:0 wei data:0xb8b...4f1a0 logs:0 hash:0xc83...37e42

status	0x0 Transaction mined but execution failed
transaction hash	0xc83bf19a4eda4898d692c1250101af53e504fe97109ea7666de9bb39da237e42
from	0x4b0897b0513fdc7c541b6d9d7e929c4e5364d2db
to	HouseLeasing.signContract() 0xbbf289d846208c16edc8474705c748aff07732db
gas	30000000 gas
transaction cost	22073 gas
execution cost	801 gas
hash	0xc83bf19a4eda4898d692c1250101af53e504fe97109ea7666de9bb39da237e42

Figure 5.6: A non-authorized user attempts to sign the smart contract.

The same policy for access restriction applies to all of the functions mentioned in Table 5.2. Our PoC implementation in Ethereum fulfills all of the requirements that have been proposed for both leasing operation and the RBAC mechanism. Conversely, the RBAC in Hyperledger Fabric works with the issuance of participant identities by the network administrator. Figure 5.7 illustrates the participant identities that have been issued by the administrator.

LANDLORD1	*In my wallet*
LANDLORD2	*In my wallet*
TENANT1	*In my wallet*
TENANT2	*In my wallet*

Figure 5.7: Participant identities issued by the administrator.

Initially, a potential tenant is assigned a contract once it has been created. Ideally, another tenant should not have access to that contract as stipulated by Table 5.2 and the access rule `R3b_PayRent{}` allows participants who are have designated access to a contract to pay a monthly rent. Figure 5.8 illustrates what happens when `TENANT1` attempts to access a contract assigned to `TENANT2` and issue the `PayRent()` function.

Submit Transaction

Transaction Type: payRent

JSON Data Preview

```
1  {
2    "$class": "estate.payRent",
3    "landlord": "resource:estate.Landlord#L1",
4    "tenant": "resource:estate.Tenant#T1",
5    "cont_terms": "resource:estate.ContractTerms#CT1",
6    "landlordID": "L1",
7    "amount": 200
8  }
```

Optional Properties

t: Participant 'estate.Tenant#T1' does not have 'CREATE' access to resource 'estate.payRent#e85a366b-183c-4f15-989e-696c69704246'

Just need quick test data? Generate Random Data | Cancel | Submit

Figure 5.8: Tenant attempts to access a contract assigned to another tenant.

Similarly, all the access control policies operate in the same manner. Hyperledger Fabric literally checks the identity of the participant that has issued an operation to the blockchain and acts according to the access control list rules implemented in the business network.

5.2.3 Negotiation Library

Our main contribution in this thesis was the design of a library that handles negotiations between the two parties in the smart contract automatically. Thus, the evaluation of the library was also our priority. We conducted tests where the two parties initiate the negotiation process by proposing new terms, reviewing the terms that have been proposed and the accept of decline the final offer. Figure 5.9 shows the terms negotiation between two participants using Ethereum blockchain.

[vm] **from:**0x147...c160c **to:**HouseLeasing.proposeTerms(uint256,uint256,uint256) 0xbbf...732db **value:**0 wei **data:**0xd77...0001c **logs:**1 **hash:**0x581...3683f

status	0x1 Transaction mined and execution succeed
transaction hash	0x581058e6d24e035a576a6116b30dba169545c2ddf7de80058aa6deef41f3683f
from	0x14723a09acff6d2a60dcdf7aa4aff308fddc160c
to	HouseLeasing.proposeTerms(uint256,uint256,uint256) 0xbbf289d846208c16edc8474705c748aff07732db
gas	30000000 gas
transaction cost	111968 gas
execution cost	89992 gas
hash	0x581058e6d24e035a576a6116b30dba169545c2ddf7de80058aa6deef41f3683f

[vm] **from:**0xca3...a733c **to:**HouseLeasing.readNewTerms() 0xbbf...732db **value:**0 wei **data:**0x8ac...79106 **logs:**1 **hash:**0x643...cce58

status	0x1 Transaction mined and execution succeed
transaction hash	0x64317f6eb20cdbe64a4d859c4f3d5977a0d8173058b463ed8a69d3132e7cce58
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	HouseLeasing.readNewTerms() 0xbbf289d846208c16edc8474705c748aff07732db
gas	30000000 gas
transaction cost	26248 gas
execution cost	4976 gas
hash	0x64317f6eb20cdbe64a4d859c4f3d5977a0d8173058b463ed8a69d3132e7cce58

[vm] **from:**0xca3...a733c **to:**HouseLeasing.acceptNewTerms() 0xbbf...732db **value:**0 wei **data:**0xc01...2d10e **logs:**1 **hash:**0x8aa...c1319

status	0x1 Transaction mined and execution succeed
transaction hash	0x8aa3a85654a7a64565a334c30676bace24b0e290637928bc1b7979a2551c1319
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	HouseLeasing.acceptNewTerms() 0xbbf289d846208c16edc8474705c748aff07732db
gas	30000000 gas
transaction cost	29633 gas
execution cost	8361 gas
hash	0x8aa3a85654a7a64565a334c30676bace24b0e290637928bc1b7979a2551c1319

Figure 5.9: Terms negotiation between the landlord and potential tenant.

Following the previous scenario, where the landlord (`0xca3...a733c`) deployed the smart contract and a potential tenant (`0x147...c160c`) opened the terms, in this example the second tries to propose new terms. Thereafter, the landlord opened the proposal made by the other participant and decided to accept the proposed terms.

During the negotiation process, the two participants communicate only through the smart contract, and the smart contract uses the library to handle the proposals. This feature allows developers to create libraries that can handle a specific workload of multiple smart contracts. Thus, generic libraries can be deployed once on the blockchain and then can be executed numerous times as reusable code, providing also anonymity throughout the smart contracts.

Similarly, the negotiation mechanism in Hyperledger Fabric follows the bilateral negotiation of the terms of contained in a smart contract. Figure 5.10 shows a tenant querying the blockchain to view the terms contained in the contract. At this point, the tenant can either accept or otherwise propose new terms and update the contract. The landlord receives an event notification of the tenant's action and can subsequently accept these new terms or counter-propose other terms.

Submit Transaction

Transaction Type negotiateTerms

JSON Data Preview


```
{
  "$class": "estate.negotiateTerms",
  "contractTermID": "CT1",
  "contid": "resource:estate.ContractTerms#CT1"
}
```


Optional Properties

Just need quick test data? Generate Random Data Cancel Submit

estate.ContractTerms

JSON Data Preview


```
{
  "$class": "estate.ContractTerms",
  "contractTermID": "CT1",
  "rent": 200,
  "penalty": 10,
  "security_deposit": 10,
  "lease_termination_amount": 10,
  "contract": "resource:estate.Contract#C1"
}
```


Optional Properties

Cancel Update

Figure 5.10: Terms negotiation transaction in Hyperledger Fabric.

5.2.3.1 Wallet Recovery

Ethereum platform supports the notion of wallets connected to the participants of the network, to keep and transfer the digital asset, ether. Therefore, the wallet recovery mechanism is only necessary for the PoC implementation in Ethereum blockchain. Figure 5.10 shows the wallet recovery process for the landlord's wallet.

The smart contract supports also the wallet recovery for the tenant's account.

[call] from:0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to:HouseLeasing.getRecoveryKeyLandlord() data:0x550...41565

transaction hash	0xbc7edad85593aacd5c859423b275e1823b19714ca74517c831694df02a72a564
from	0xca35b7d915458ef540ade6068dfe2f44e8fa733c
to	HouseLeasing.getRecoveryKeyLandlord() 0xbbf289d846208c16edc8474705c748aff07732db
transaction cost	21840 gas (Cost only applies when called by a contract)
execution cost	568 gas (Cost only applies when called by a contract)
hash	0xbc7edad85593aacd5c859423b275e1823b19714ca74517c831694df02a72a564
decoded output	{ "0": "uint256: 112921747135283690483540515570353485096365546789862756717150072927109406082735" }

[vm] from:0x583...40225 to:HouseLeasing.recoverLandlord(uint256) 0xbbf...732db
value:0 wei data:0xede...d46af logs:1 hash:0x6ec...48025

status	0x1 Transaction mined and execution succeed
transaction hash	0x6ec87d533c3e053b31e1848ecf896e829d09c3d49ff7786e53c0397b24d48025
from	0x583031d1113ad414f02576bd6afabfb302140225
to	HouseLeasing.recoverLandlord(uint256) 0xbbf289d846208c16edc8474705c748aff07732db
gas	30000000 gas
transaction cost	31126 gas
execution cost	7678 gas
hash	0x6ec87d533c3e053b31e1848ecf896e829d09c3d49ff7786e53c0397b24d48025

Figure 5.11: Landlord links a new wallet to the smart contract.

The first transaction is an automated transaction initiated by the smart contract, once it is deployed on the blockchain, and produces a unique recovery code for the landlord. The landlord keeps safely this number and can use it at anytime to link another wallet to the smart contract. The second transaction is made by the landlord using another Ethereum wallet address and the recovery code obtained after the creation of the contract. This method provides some level of security towards exploits and attacks. Guessing the unique recovery code is a very hard process, because of Ethereum's built-in currency and the notion of gas.

5.3 Ethereum gas consumption

Although we mentioned we will not evaluate the system in terms of performance, we conducted a performance evaluation for Ethereum in terms of gas consumption. We performed multiple executions of the smart contract and the functions and we measured the gas consumption of each one individually. Table 5.3 lists the smart contract functions used for leasing and negotiation process respectively, along with the gas consumption.

Function	Transaction Cost	Execution Cost
`HouseLeasing()`	2,211,891 wei	1,682,031 wei
`NegotiationLibrary()`	125,344 wei	54,552 wei
`readTerms()`	92,109 wei	70,837 wei
`signContract()`	28,686 wei	7,414 wei
`payRent()`	152,642 wei	130,922 wei
`terminate()`	29,491 wei	8,219 wei
`proposeTerms()`	111,968 wei	89,992 wei
`readNewTerms()`	26,248 wei	4,976 wei
`acceptNewTerms()`	29,633 wei	8,361 wei
`declineNewTerms()`	29,578 wei	8,306 wei

Table 5.3: Ethereum performance evaluation in terms of gas consumption.

In the table, the transaction cost is based on the cost of sending data to the blockchain and depends on four parameters: the base cost of a transaction, the cost of a contract deployment, the cost for every zero byte of data or code for a transaction, and the cost of every non-zero byte of data or code for a transaction. However, the execution cost is based on the cost of computational operations which are executed as a result of the transaction. Figure 5.11 shows Ethereum price over the past two years which can be related to the gas consumption measured in wei. Ethereum price hit a record high of $1,427.05 in 13th of January 2018 [47].

Figure 5.12: Ethereum price in relation to USD and Bitcoin [47].

In relation, to our PoC implementation in Ethereum and our gas consumption evaluation, we also calculated the total cost of maintaining the smart contract and the library for a year. The cost of deploying the smart contract (and the library), signing the agreement and receive 12 consecutive payments within a year will cost about 4,3 billion wei or $4.3 * 10^{-12}$ ether. Therefore, for a company that owns 1,000,000 properties and uses our PoC implementation in Ethereum as their leasing tool, will cost about 0.00000429 ether or $0.0061 for all transactions.

6 Conclusion

In this chapter, we conclude the thesis by first discussing the results of the evaluation process and secondly by proposing elements of interest for future work.

6.1 Discussion

In this thesis, a comparative analysis was conducted between private and public blockchain technologies to understand the architectural differences and similarities. Specifically, we looked at the manner in which the two blockchain technologies handle data privacy. From our PoC implementation, we can note that the blockchain has inbuilt capabilities to guarantee data ownership by way of limiting access control to smart contract contents, however, careful consideration and due diligence must be conducted on the choice of blockchain technology before any target application is built. This is basically a way of maximizing the benefits that come with the use of blockchain technology.

Our other objective in this thesis was to design a mechanism for negotiation of the terms of a smart contract. Although smart contracts offer many advantages such as security, transparency, and efficiency they lack the flexibility that comes with paper-based contracts in relation to term negotiation. We evaluated our mechanism and found it to meet all the requirements of the evaluation process. From the evaluation of the negotiation mechanism, we can note that, although the negotiation process is somewhat tedious and long, it can be eased using the benefits of the blockchain technology. In addition, our negotiation scheme (generic library) can find use in many different applications aside from the commercial real estate.

We also looked at the performance, Ethereum smart contract's deployment and operation seem very inexpensive process, thus it can be viewed as a potential candidate for developing other types of agreement as well. Although Ethereum blockchain seems an affordable panacea for all kind of applications, it introduces multiple vulnerabilities into the system due to its permissionless type of operation. For that reason, we also designed a wallet recovery mechanism where the users registered to the smart contract can securely restore their account if they lost access to it.

The research questions posed in this thesis have all been answered by the theory presented in our work and further exemplified by the design and subsequent implementation of our PoC. The reader can find a summary of our research questions below:

- **Question 1: What is the main difference between public and private blockchain regarding the access restriction to the contents of the smart contract?**
- **Question 2: Can we design a library to achieve negotiation of smart contract terms and evaluate the library using a use case implementation?**
- **Question 3: Can we implement a role-based access control to smart contract contents based on the source and destination ID of the transaction?**
- **Question 4: If so, what are the implications on the scalability of such an implementation in Ethereum, a public blockchain?**

Research Question 1 is answered in Section 3.3 of Chapter 3 that presented the similarities as well as the differences laid down in Table 3.1. Partially this research question has also been answered in Chapter 2. Research Question 2 has been answered in Chapters 4 and 5, which presents the implementation and evaluation of the negotiation mechanism respectively. Similarly, Research Question 3 has been answered in Chapters 4 and 5. Finally, Research Question 4 has been answered in Chapter 5 that presents the performance in terms of ether exhaustion.

The blockchain is a promising technology that will revolutionize how business houses transact. The comparative analysis in this thesis has shown the applicability of the blockchain technology in today's ever changing business world. However, it should be noted that a clear understanding of the business requirements is needed before embarking on the journey to adopt the blockchain. Furthermore, we have demonstrated that it is possible to facilitate a bi-directional negotiating scheme of the many contract terms that govern a business transaction. The blockchain has in-built capabilities to guarantee data ownership with the private blockchain having a matured ecosystem in this regard. Lastly, designing scalable solutions on the Etherieum blockchain is imperative, due to the fact that every computation consumes an amount of ether that must be recovered by the miners.

6.2 Future Work

Blockchain technology is at the moment one of the hot topics and currently undergoing constant evolution. Concentrated interest from both industry and academia only leaves room for further research. It is our understanding that the novelty nature of

the blockchain technology calls for more technological development. There is a need for a fine-grained understanding of the technology and the problem domain that it attempts to solve. In the context of our thesis, there are elements of interest that can be further enhanced. For instance, a mechanism to further verify that the terms of the contract have indeed been met so that the parties of the contract are satisfied, is necessary. In addition, our current version of the negotiation scheme only allows a two-party negotiating scheme. In the future, it is imperative that a multi-party negotiation scheme is developed to allow more players to negotiate. Furthermore, there is also a huge potential for research on the need to verify the code contained in the contract using formal methods. Lastly, we believe there is a need for evaluation using testing. This kind of evaluation, can include functional testing and structural testing.

Bibliography

[1] Satoshi Nakamoto, “Bitcoin: A Peer-to-Peer Electronic Cash System”, 2008. Available at: `https://bitcoin.org/bitcoin.pdf`

[2] Jerry Brito and Andrea Castillo, “Bitcoin: A Primer for Policymakers”, 2013. Available at: `https://www.mercatus.org/system/files/Brito_BitcoinPrimer.pdf`

[3] Tien Tuan Anh Dinh, Rui Liu, Meihui Zhang, “Untangling Blockchain: A Data Processing View of Blockchain Systems”, in *Proceedings of the IEEE Transactions on Knowledge and Data Engineering*, 2017.

[4] Vitalik Buterin, “A Next-Generation Smart Contract and Decentralized Application Platform”, 2013. Available at: `https://github.com/ethereum/wiki/wiki/White-Paper`

[5] Christian Cachin, IBM Research, Zurich, “Architecture of the Hyperledger Blockchain Fabric”, 2016. Available at: `https://www.zurich.ibm.com/dccl/papers/cachin_dccl.pdf`

[6] David Schwartz, Noah Youngs, Arthur Britto, Ripple Labs Inc, “The Ripple Protocol Consensus Algorithm”, 2014. Available at: `https://ripple.com/files/ripple_consensus_whitepaper.pdf`

[7] Coin Central, “Coin Central”, 2018. Available at: `https:https://coincentral.com/how-many-bitcoins-are-left/`

[8] Marko Vukolic, “The Quest for Scalable Blockchain Fabric: Proof-of-Work vs. BFT Replication”, in *Problems in Network Security, Proc. IFIP WG 11.4 Workshop*, 2015.

[9] Martin Valenta and Philipp Sandner, “Comparison of Ethereum, Hyperledger Fabric and Corda”, in *FSBC Working Paper, Frankfurt School: Blockchain Center*, 2017.

[10] David Remnick, “Cambridge Analytica and a Moral Reckoning in Silicon Valley”, 2018. Available at: `https://www.newyorker.com/magazine/2018/04/02/cambridge-analytica-and-a-moral-reckoning-in-silicon-valley`

[11] Nick Szabo, “Smart Contracts: Building Blocks for Digital Markets”, 1996. Available at: http://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart_contracts_2.html

[12] Goldman Sachs Group, “Blockchain: putting theory into practice”, 2016. Available at: https://www.unlock-bc.com/sites/default/files/attachments/Goldman-Sachs-report-Blockchain-Putting-Theory-into-Practice.pdf

[13] Smart Contracts Alliance and Nick Szabo “Smart Contracts: 12 Use Cases for Business and Beyond”, 2016. Available at: https://www.bloq.com/assets/smart-contracts-white-paper.pdf

[14] Deloitte Centre for Financial Services, “Blockchain in commercial real estate the future is here”, 2017. Available at: https://www2.deloitte.com/content/dam/Deloitte/us/Documents/financial-services/us-dcfs-blockchain-in-cre-the-future-is-here.pdf

[15] Damien Cosset, “Blockchain: What is Mining?”, 2018. Available at: https://dev.to/damcosset/blockchain-what-is-mining-2eod/

[16] Victoria McIntosh, Information & Privacy Professional, “Blockchain and Privacy: the New Frontier”, 2017. Available at: https://victoriamcintosh.com/privacy-vs-blockchain/

[17] Gregory Rocco, “Blockchain Technology and Data: Identity, Storage, Exchange”, 2018. Available at: https://www.ccn.com/blockchain-technology-and-data-identity-storage-exchange/

[18] Karl Wüst and Arthur Gervais , “Do You Need a Blockchain?”, ePrint Archive, Report 375, 2017. Available at: https://eprint.iacr.org/2017/375/

[19] Chad Decker, “Ethereum vs. Hyperledger”, 2017. Available at: https://blockchaintrainingalliance.com/blogs/news/ethereum-vs-hyperledger

[20] S. Pongnumkul, C. Siripanpornchana and S. Thajchayapong, “Performance Analysis of Private Blockchain Platforms in Varying Workloads”, in *Proceedings of the 26th International Conference on Computer Communication and Networks (ICCCN)*, 2017.

[21] Damiano Di Francesco Maesa, Paolo Mori, and Laura Ricci, “Blockchain Based Access Control”, in Proceedings of *IFIP International Conference on Distributed Applications and Interoperable Systems, Neuchatel, Switzerland*, 2017.

[22] S. Thomas, IETF Draft “Crypto-Conditions”, 2017. Available at: https://tools.ietf.org/html/draft-thomas-crypto-conditions-03

[23] OpenPGP. Available at: https://www.openpgp.org/

[24] Neal H. Walfield, "An Advanced Introduction to GnuPG", 2017. Available at: https://gnupg.org/ftp/people/neal/an-advanced-introduction-to-gnupg/openpgp/openpgp.pdf

[25] R. Charette, "DigiNotar certificate authority breach crashes e-government in The Netherlands", *IEEE Spectr.*, 2011. Available at: https://spectrum.ieee.org/riskfactor/telecom/security/diginotar-certificate-authority-breach-crashes-egovernment-in-the-netherlands

[26] Vincenzo Scoca, Rafael Brundo Uriarte, Rocco De Nicola, "Smart Contract Negotiation in Cloud Computing", in *Proceedings of the IEEE 10th International Conference on Cloud Computing (CLOUD)*, 2017.

[27] Aafaf Ouaddah, Anas Abou Elkalam, Abdellah Ait Ouahman, "FairAccess: a new Blockchain-based access control framework for the Internet of Things", in *Proceedings of Security and Communication Networks*, vol. 9, no. 18, pp. 5943–5964, 2017.

[28] Yuanyu Zhang, Shoji Kasahara, Yulong Shen, Xiaohong Jiang, Jianxiong Wan, "Smart Contract-Based Access Control for the Internet of Things", 2018. Available at: https://arxiv.org/pdf/1802.04410.pdf

[29] J. P. Cruz and Y. Kaji, "The bitcoin network as platform for trans- organizational attribute authentication", in *IPSJ Trans. Math. Model. Appl., vol. 9, no. 2, pp. 41–48*, 2016.

[30] Ronald L. Rivest, "Cryptography", In J. Van Leeuwen, Handbook of Theoretical Computer Science, 1, Elsevier, 1990.

[31] Alfred J. Menezes, Paul C. van Oorschot, Scott A. Vanstone, "Introduction to Modern Cryptography". p. 10, 2005.

[32] Mehrdad S. Sharbaf, "Quantum cryptography: An emerging technology in network security", in *Proceedings of the 2011 IEEE International Conference on Technologies for Homeland Security (HST): 13–19*, 2011.

[33] R. L. Rivest, "The MD5 message-digest algorithm", Internet Requests for Comments, RFC Editor, RFC 1321, 1992. Available at: http://www.rfc-editor.org/rfc/rfc1321.txt

[34] Xiaoyun Wang, Hongbo Yu, "How to Break MD5 and Other Hash Functions", Advances in Cryptology – Lecture Notes in Computer Science, pp. 19–35, 2005.

[35] D. Eastlake and P. Jones, "US Secure Hash Algorithm 1 (SHA1)", Internet Requests for Comments, RFC Editor, RFC 3174, 2001. Available at: http://www.rfc-editor.org/rfc/rfc3174.txt

[36] Microsoft, "Windows Enforcement of Authenticode Code Signing and Timestamping", 2015.

[37] Google, "Intent to Deprecate: SHA-1 certificates", 2014.

[38] Apple Inc., "Safari and WebKit ending support for SHA-1 certificates – Apple Support", 2017.

[39] Mozilla, "CA:Problematic Practices – MozillaWiki", 2014.

[40] B. Preneel, "The first 30 years of cryptographic hash functions and the NIST SHA-3 competition", in Cryptographers' Track at the RSA Conference, Springer, pp. 1–14, 2010.

[41] CWI Amsterdam and Google Reseach, Marc Stevens, Elie Bursztein, Pierre Karpman, Ange Albertini, Yarik Markov, "The first collision for full SHA-1", 2017.

[42] W. Diffie and M. Hellman, "New directions in cryptography", in *Proceedings of IEEE transactions on Information Theory, vol. 22, no. 6, pp. 644–654*, 1976.

[43] J. Nechvatal, "Public-key cryptography", DTIC Document, Tech. Rep., 1991. Available: `https://pdfs.semanticscholar.org/720f/1f0cf8699d94ab775d0da44ebef74450de7f.pdf`

[44] R. Rivest, A. Shamir, and L. Adleman, "A method for obtaining digital signatures and public-key cryptosystems", in CACM, 21(2), 120–126, 1978.

[45] M. Ali, J. Nelson, R. Shea and M. J. Freedman, "Blockstack: A Global Naming and Storage System Secured by Blockchains", in *Proceedings of the 2016 USENIX Annual Technical Conference, Denver, CO*, 2016.

[46] A. Hevner, S. March, J. Park, S. Ram, "Design Science in Information Systems Research", MIS Quarterly 28, 75–105, 2004.

[47] CoinMarketCap: Cryptocurrency Market Capitalizations, 2018.
Available at: `https://coinmarketcap.com/`

A

Appendix A

Ethereum implementation consists of two source files placed in the same directory:

- HouseLeasing.sol
- NegotiationLibrary.sol

The source code of HouseLeasing.sol is listed below:

```
pragma solidity ^0.4.21;

/* Import library for negotiations process */
import {NegotiationLibrary} from "./NegotiationLibrary.sol";

contract HouseLeasing {

    /* Use library for negotiations process */
    using NegotiationLibrary for NegotiationLibrary.Proposal;
    NegotiationLibrary.Proposal proposal;

    /* Declaration of type which holds the rents paid */
    struct RentPaid {
        uint paymentID; /* The payment id */
        uint paymentDate; /* The date of the payment */
        uint rent; /* The amount of rent that is paid */
        uint month;
        uint year;
    }

    /* List of the rents that have been paid */
    RentPaid[] private rentspaid;

    /* Participants */
    address private landlord;
    address private tenant;

```

```
28      /* Address of the smart contract */
29      address thisAddress = this;
30
31      /* House details */
32      string private houseAddress;
33      string private houseZipCode;
34      string private houseCity;
35      string private termsText = "[House details] => [1. Address, 2. Zip code,
36      3. City, 4. Landlord's Ethereum address] , [Leasing terms] => [5. Rent
37      (in wei), 6. Leasing duration (in months), 7. Last calendar day to pay,
38      8. Contract created (timestamp)]";
39
40      /* Leasing terms */
41      uint private contractCreated;
42      uint private rent;
43      uint private duration;
44      uint private lastDay;
45
46      /* Recovery keys */
47      uint private recoveryKeyTenant = 0;
48      uint private recoveryKeyLandlord = 0;
49
50      /* State of the contract */
51      enum State {Created, Initiated, Terminated, underReview}
52      State private state;
53
54      /* State of the negotiation process */
55      enum NegotiationState {landlordProposed, tenantProposed}
56      NegotiationState private negotiationState;
57
58      /* Initiation of the contract by the landlord */
59      function HouseLeasing(string Address, string ZipCode, string City, uint
60      Rent, uint Duration, uint LastDayToPay) public {
61          houseAddress = Address;
62          houseZipCode = ZipCode;
63          houseCity = City;
64          rent = Rent;
65          duration = Duration;
66          lastDay = LastDayToPay;
67          landlord = msg.sender;
68          contractCreated = block.timestamp;
69          negotiationState = NegotiationState.landlordProposed;
70          recoveryKeyLandlord = uint(keccak256(block.difficulty, now));
71      }
72
73
```

```
74      /* Function modifiers for restricting access */
75      modifier _require(bool _condition) {
76          if (!_condition) revert();
77          _;
78      }
79      modifier onlyLandlord() {
80          if (msg.sender != landlord) revert();
81          _;
82      }
83      modifier onlyTenant() {
84          if (msg.sender != tenant) revert();
85          _;
86      }
87      modifier isState(State _state) {
88          if (state != _state) revert();
89          _;
90      }
91      modifier isNegotiationState(NegotiationState _negotiationState) {
92          if (negotiationState != _negotiationState) revert();
93          _;
94      }
95
96      /* Helper functions to read the values from the blockchain at any time */
97      function getRentPaid(uint id) public view returns (uint, uint, uint, uint,
98      uint) {
99          return (rentspaid[id].paymentID, rentspaid[id].paymentDate,
100         rentspaid[id].rent, rentspaid[id].month, rentspaid[id].year);
101     }
102     function getHouseDetails() public view returns (string, string, string) {
103         return (houseAddress, houseZipCode, houseCity);
104     }
105     function getLandlord() public view returns (address) {
106         return landlord;
107     }
108     function getTenant() public view returns (address) {
109         return tenant;
110     }
111     function getContractCreated() public view returns (uint) {
112         return contractCreated;
113     }
114     function getContractAddress() public view returns (address) {
115         return thisAddress;
116     }
117     function getRecoveryKeyTenant() public view returns (uint) {
118         return recoveryKeyTenant;
119     }
```

```
120     function getRecoveryKeyLandlord() public view returns (uint) {
121         return recoveryKeyLandlord;
122     }
123     function getState() public view returns (string) {
124         if (state == State.Created){
125             return "Created";
126         }
127         if (state == State.Initiated){
128             return "Initiated";
129         }
130         if (state == State.Terminated){
131             return "Terminated";
132         }
133         if (state == State.underReview){
134             return "underReview";
135         }
136     }
137     function getNegotiationState() public view returns (string) {
138         if (negotiationState == NegotiationState.landlordProposed){
139             return "landlordProposed";
140         }
141         if (negotiationState == NegotiationState.tenantProposed){
142             return "tenantProposed";
143         }
144     }
145     function getBalance() public view returns(uint) {
146         return msg.sender.balance;
147     }
148
149     /* Events for the dApp */
150     event termsRead();
151     event contractSigned();
152     event rentPaid();
153     event contractTerminated();
154
155     /* Read the terms of the contract - no input required */
156     function readTerms() public
157     _require(state == State.Created || ( (state == State.underReview ||
158     state == State.Initiated) && (msg.sender == landlord || msg.sender ==
159     tenant) ) )
160     returns (string, string, string, string, address, uint, uint, uint, uint)
161     {
162         emit termsRead();
163         tenant = msg.sender;
164         state = State.underReview;
165         recoveryKeyTenant = uint(keccak256(block.difficulty, now));
```

```
166         return (termsText, houseAddress, houseZipCode, houseCity, landlord,
167         rent, duration, lastDay, contractCreated);
168     }
169 
170     /* Sign the contract as tenant - no input required */
171     function signContract() public
172     onlyTenant
173     isState(State.underReview)
174     _require(msg.sender != landlord)
175     {
176         emit contractSigned();
177         state = State.Initiated;
178     }
179 
180     /* Pay the rent as tenant - takes as input two integers (month, year)
181                                 - and as a value the amount of the rent */
182     function payRent(uint month, uint year) public payable
183     onlyTenant
184     isState(State.Initiated)
185     _require(msg.value == rent)
186     {
187         emit rentPaid();
188         landlord.transfer(msg.value);
189         rentspaid.push(RentPaid({paymentID:(year*100)+month,
190         paymentDate:block.timestamp, rent:msg.value, month:month, year:year}));
191     }
192 
193     /* Terminate the contract so the tenant can't pay rent anymore */
194     function terminate() public
195     onlyLandlord
196     {
197         emit contractTerminated();
198         /* If there is any value on the contract send it to the landlord */
199         landlord.transfer(thisAddress.balance);
200         state = State.Terminated;
201     }
202 
203     /* Events for the DApps*/
204     event termsRequested();
205     event newTermsRead();
206     event termsAccepted();
207     event termsDeclined();
208 
209     /* Propose new terms - takes as input three integers (rent, duration,
210     last day to pay) */
211     function proposeTerms(uint newRent, uint newDuration, uint newLastDay) public
```

```
212     isState(State.underReview)
213     _require(msg.sender == tenant || msg.sender == landlord)
214     {
215         emit termsRequested();
216         if(msg.sender == tenant && negotiationState ==
217         NegotiationState.landlordProposed){
218             negotiationState = NegotiationState.tenantProposed;
219             proposal.makeProposal(newRent, newDuration, newLastDay);
220         }
221         if(msg.sender == landlord && negotiationState ==
222         NegotiationState.tenantProposed ){
223             negotiationState = NegotiationState.landlordProposed;
224             proposal.makeProposal(newRent, newDuration, newLastDay);
225         }
226     }
227
228     /* Read the terms of the contract that have been proposed - no input
229     required */
230     function readNewTerms() public
231     isState(State.underReview)
232     _require(msg.sender == tenant || msg.sender == landlord)
233     returns (uint, uint, uint, uint)
234     {
235         emit newTermsRead();
236         return proposal.readProposal();
237     }
238
239     /* Accept the terms of the contract that have been proposed - no input
240     required */
241     function acceptNewTerms() public
242     isState(State.underReview)
243     _require(msg.sender == tenant || msg.sender == landlord)
244     {
245         emit termsAccepted();
246         if(msg.sender == tenant && negotiationState ==
247         NegotiationState.landlordProposed){
248             state = State.Initiated;
249         }
250         if(msg.sender == landlord && negotiationState ==
251         NegotiationState.tenantProposed ){
252             state = State.Initiated;
253         }
254     }
255
256
257
```

```
258     /* Decline the terms of the contract that have been proposed - no input
259     required */
260     function declineNewTerms() public
261     isState(State.underReview)
262     _require(msg.sender == tenant || msg.sender == landlord)
263     {
264         emit termsDeclined();
265         if(msg.sender == tenant && negotiationState ==
266         NegotiationState.landlordProposed){
267             state = State.Terminated;
268         }
269         if(msg.sender == landlord && negotiationState ==
270         NegotiationState.tenantProposed ){
271             state = State.Terminated;
272         }
273     }
274 
275     /* Events for the DApps*/
276     event landlordWalletLinked();
277     event tenantWalletLinked();
278 
279     /* Link a new wallet to the smart contract (only for landlord) - takes as
280     input the recovery key */
281     function recoverLandlord(uint key) public
282     _require(state == State.Created || state == State.Initiated ||
283     state == State.underReview)
284     {
285         emit landlordWalletLinked();
286         if(key == recoveryKeyLandlord){
287             landlord = msg.sender;
288         }
289     }
290 
291     /* Link a new wallet to the smart contract (only for tenant) - takes as
292     input the recovery key */
293     function recoverTenant(uint key) public
294     _require(state == State.Initiated || state == State.underReview)
295     {
296         emit tenantWalletLinked();
297         if(key == recoveryKeyTenant){
298             tenant = msg.sender;
299         }
300     }
301 }
```

The source code of `NegotiationLibrary.sol` is listed below:

```
pragma solidity ^0.4.21;

library NegotiationLibrary {

    /* Declaration of type which holds the proposals */
    struct Proposal {
        uint rentProposed;
        uint durationProposed;
        uint lastDayProposed;
        uint dateProposed;
    }

    /* Propose new terms (rent, duration, last calendar day to pay) */
    function makeProposal(Proposal storage proposals, uint newRent,
    uint newDuration, uint newLastDay) public {
        proposals.rentProposed = newRent;
        proposals.durationProposed = newDuration;
        proposals.lastDayProposed = newLastDay;
        proposals.dateProposed = block.timestamp;
    }

    /* Read the proposed terms */
    function readProposal(Proposal storage proposals) public view
    returns (uint, uint, uint, uint) {
        return (proposals.rentProposed, proposals.durationProposed,
        proposals.lastDayProposed, proposals.dateProposed);
    }
}
```

B

Appendix B

Hyperledger Fabric implementation consists of two source files placed in the same directory:

- `estate.cto`
- `logic.js`
- `permissions.acl`

Hyperledger Composer includes an object-oriented modeling language that is used to define the domain model for a business network definition. The Hyperledger Composer CTO file is composed of the following elements:

- A single namespace where all resource declarations within the file are implicitly in this namespace.
- A set of resource definitions, encompassing assets, transactions, participants, and events.
- Optional import declarations that import resources from other namespaces.

The source code of `estate.cto` is listed below:

```
1  namespace estate
2  
3  enum ContractState {
4    o CREATED
5    o INITIATED
6    o TERMINATED
7    o UNDER_REVIEW
8  }
9  
10 asset House identified by houseID {
11   o String houseID
12   --> Landlord landlord
13   o String information
```

```
14    o Boolean forRent optional
15  }
16
17  asset Contract identified by contractID{
18    o String contractID
19    --> Landlord landlord
20    --> Tenant tenant
21    --> House house
22    o ContractState status
23  }
24
25  asset ContractTerms identified by contractTermID{
26    o String contractTermID
27    o Double rent
28    o Double penalty
29    o Double security_deposit
30    o Double lease_termination_amount
31    --> Contract contract
32  }
33
34  participant Landlord identified by landlordID {
35    o String landlordID
36    o String name
37    o Double balance
38
39  }
40
41  participant Tenant identified by tenantID {
42    o String tenantID
43    o String name
44    o Double balance
45  }
46
47  transaction payRent {
48    --> Landlord landlord
49    --> Tenant tenant
50    --> ContractTerms cont_terms
51    o String landlordID
52    o Double amount
53  }
54
55  transaction createHouse{
56    o String houseID
57    o String information
58    --> Landlord landlord
59  }
```

```
60
61 transaction createLandlord{
62   o String landlordID
63   o String name
64   o Double balance
65 }
66
67 transaction createTenant{
68   o String tenantID
69   o String name
70   o Double balance
71   o Boolean forRent
72 }
73
74 transaction createContract{
75   o String contractID
76   --> Landlord landlord
77   --> Tenant tenant
78   --> House house
79   o ContractState status
80 }
81
82 transaction createTerms{
83   o String contractTermID
84   o Double rent
85   o Double penalty
86   o Double security_deposit
87   o Double lease_termination_amount
88   --> Contract contract
89 }
90
91 transaction negotiateTerms{
92   o String contractTermID
93   --> ContractTerms contid
94 }
```

The main application in Hyperledger Fabric is written in JavaScript and runs within the `Node.js` platform, allowing users to create, read, update and delete assets and participants, while also to submit transactions across the business network.

The source code of `logic.js` is listed below:

```
'use strict';

/**
 * Transaction to create contract
 * @param {estate.createContract} cont
 * @transaction
 */
async function createContract(cont) {
  const factory = getFactory();
  const NS = 'estate';
  const contractID = cont.contractID;
  const tenantID = cont.tenant.tenantID;
  const landlordID = cont.landlord.landlordID;
  const houseID = cont.house.houseID;

  // Create a contract between a tenant and a landlord
  const contractRegistry = await getAssetRegistry(NS + '.Contract');
  const contract = factory.newResource(NS, 'Contract', contractID);
  // Set the properties of the contract
  contract.contractID = cont.contractID;
  contract.landlord = factory.newRelationship(NS, 'Landlord', landlordID);
  contract.tenant = factory.newRelationship(NS, 'Tenant', tenantID);
  contract.house = factory.newRelationship(NS, 'House', houseID);
  contract.status = 'CREATED';
  await contractRegistry.add(contract);
}

/**
 * Transaction to create the house
 * @param {estate.createHouse} cont
 * @transaction
 */
async function createHouse(cont) {
  const factory = getFactory();
  const NS = 'estate';
  const houseID = cont.houseID;
  const landlordID = cont.landlord.landlordID;

  // Create the house
  const houseRegistry = await getAssetRegistry(NS + '.House');
```

```
41    const house = factory.newResource(NS, 'House', houseID);
42    // Set the house properties
43    house.houseID = cont.houseID;
44    house.information = cont.information;
45    house.landlord = factory.newRelationship(NS, 'Landlord', landlordID);
46    await houseRegistry.add(house);
47  }
48
49  /**
50   * Transaction to create the tenant
51   * @param {estate.createTenant} cont - the house asset
52   * @transaction
53   */
54  async function createTenant(cont) {
55    const factory = getFactory();
56    const NS = 'estate';
57    const tenantID = cont.tenantID;
58
59    // Create the tenant
60    const tenantRegistry = await getParticipantRegistry(NS + '.Tenant');
61    const tenant = factory.newResource(NS, 'Tenant', tenantID);
62    // Set the tenant's properties
63    tenant.tenantID = cont.tenantID;
64    tenant.name = cont.name;
65    tenant.balance = cont.balance;
66    await tenantRegistry.add(tenant);
67  }
68
69  /**
70   * Transaction to create the landlord
71   * @param {estate.createLandlord} cont - the house asset
72   * @transaction
73   */
74  async function createLandlord(cont) {
75    const factory = getFactory();
76    const NS = 'estate';
77    const landlordID = cont.landlordID;
78
79    // Create the landlord
80    const landlordRegistry = await getParticipantRegistry(NS + '.Landlord');
81    const landlord = factory.newResource(NS, 'Landlord', landlordID);
82    // Set the landlord's properties
83    landlord.landlordID = cont.landlordID;
84    landlord.name = cont.name;
85    landlord.balance = cont.balance;
86    await landlordRegistry.add(landlord);
```

```
87  }
88
89  /**
90   * Transaction to pay rent by the tenant
91   * @param {estate.payRent} cont
92   * @transaction
93   */
94   async function payRent(cont) {
95       const amount = cont.amount;
96       const cont_rent = cont.cont_terms.rent;
97       const ten_balance = cont.tenant.balance;
98
99       if(amount < cont_rent || amount < ten_balance){
100          console.log("Insuficient balance")
101      }
102      else{
103          // Apply the rent
104          cont.landlord.balance += cont.amount;
105          cont.tenant.balance -= cont.amount;
106
107          // Update the tenants balance
108          const tenant = await await getParticipantRegistry('estate.Tenant');
109          await tenant.update(cont.tenant);
110
111          // Update the landlords balance
112          const landlord = await await getParticipantRegistry('estate.Landlord');
113          await landlord.update(cont.landlord);
114
115          console.log("Rent of: "+amount+" has been paid");
116      }
117  }
118
119 /**
120  * Transaction to create the contract terms
121  * @param {estate.createTerms} cont
122  * @transaction
123  */
124 async function createTerms(cont) {
125   const factory = getFactory();
126   const NS = 'estate';
127   const contractTermID = cont.contractTermID;
128   const contractID = cont.contract.contractID;
129
130   // Create the terms of the contract
131   const contractTermRegistry = await getAssetRegistry(NS + '.ContractTerms');
132   const contractTerm = factory.newResource(NS, 'ContractTerms', contractTermID);
```

```
133       // Set the properties of the contract
134       contractTerm.contractTermID = cont.contractTermID;
135       contractTerm.rent = cont.rent;
136       contractTerm.penalty = cont.penalty;
137       contractTerm.security_deposit = cont.security_deposit;
138       contractTerm.lease_termination_amount = cont.lease_termination_amount;
139       contractTerm.contract = factory.newRelationship(NS, 'Contract', contractID);
140       await contractTermRegistry.add(contractTerm);
141   }
142
143   /**
144    * Transaction to create the negotiation mechanism
145    * @param {estate.negotiateTerms} cont
146    * @transaction
147    */
148   async function negotiateTerms(cont) {
149       const assetRegistry = await getAssetRegistry('estate.ContractTerms');
150       const results = await query('selectContract');
151       //const contid = cont.contid.contractTermID;
152
153       // Since all registry requests have to be serialized anyway, there
154       // is no benefit to calling Promise.all on an array of promises
155       results.forEach(async ContractTerms => {
156           const removeNotification = getFactory().newEvent('org.example.trading',
157           'RemoveNotification');
158           removeNotification.commodity = trade;
159           emit(removeNotification);
160           await assetRegistry.get(ContractTerms);
161       });
162   }
```

ACL rules are defined into `permissions.acl` file, which is located in the root of the business network. If this file is missing from the business network then all access is permitted.

The source code of `permissions.acl` is listed below:

```
/*
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *
 * http://www.apache.org/licenses/LICENSE-2.0
 *
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

rule SystemACL {
    description:  "System ACL to permit all access"
    participant: "org.hyperledger.composer.system.Participant"
    operation: ALL
    resource: "org.hyperledger.composer.system.**"
    action: ALLOW
}

rule NetworkAdminUser {
    description: "Grant business network administrators full access to user
    resources"
    participant: "org.hyperledger.composer.system.NetworkAdmin"
    operation: ALL
    resource: "**"
    action: ALLOW
}

rule NetworkAdminSystem {
    description: "Grant business network administrators full access to system
    resources"
    participant: "org.hyperledger.composer.system.NetworkAdmin"
    operation: ALL
    resource: "org.hyperledger.composer.system.**"
    action: ALLOW
}

```

```
41 rule R1b_ReadTerms {
42   description: "Tenant can read terms only if the ID is equivalent to
43   participant"
44   participant(t): "estate.Tenant"
45   operation: ALL
46   resource(c): "estate.readTerms"
47   condition: (c.owner.getIdentifier() == t.getIdentifier())
48   action: ALLOW
49 }
50
51 rule R2b_SignContract {
52   description: "Tenant can only sign a contract previously assigned to them
53   using an ID"
54   participant(t): "estate.Tenant"
55   operation: READ, UPDATE
56   resource(v): "estate.SignContract"
57   condition: (v.getIdentifier() == t.getIdentifier())
58   action: ALLOW
59 }
60
61 rule R3b_PayRent {
62   description: "Tenant can only call the PayRent function only if they have
63   their ID assigned"
64   participant(t): "estate.Tenant"
65   operation: ALL
66   resource(c): "estate.PayRent"
67   condition: (c.owner.getIdentifier() == t.getIdentifier())
68   action: ALLOW
69 }
70
71 rule R4b_Terminate{
72   description: "Landlord can only call the terminate function only if the
73   contract is assigned to them"
74   participant(t): "estate.Landlord"
75   operation: ALL
76   resource(c): "estate.Terminate"
77   condition: (c.owner.getIdentifier() == t.getIdentifier())
78   action: ALLOW
79 }
```