DEPARTMENT OF
COMPUTER SCIENCE

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

RODRIGO MIGUEL NEVES AREDE

BSc in Computer Science and Engineering

BLOCKOPOLY

A BLOCKCHAIN-BASED APPROACH TO LAND REGISTRY

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2024

NOVA SCHOOL OF
SCIENCE & TECHNOLOGY

DEPARTMENT OF
COMPUTER SCIENCE

BLOCKOPOLY

A BLOCKCHAIN-BASED APPROACH TO LAND REGISTRY

RODRIGO MIGUEL NEVES AREDE

BSc in Computer Science and Engineering

Adviser: André Simões
Product Line Manager, Crossjoin Solutions

Co-adviser: Artur Miguel Dias
Assistant Professor, NOVA University Lisbon

MASTER IN COMPUTER SCIENCE AND ENGINEERING

NOVA University Lisbon
September, 2024

Blockopoly
A Blockchain-based Approach to Land Registry

This document was created with the (pdf/Xe/Lua)LaTeX processor and the NOVAthesis template (v7.1.5) [41].

A special thanks to Nuno, for his continuous support, guidance, and encouragement throughout this journey. To the remaining team at Crossjoin, thanks to André and Sebastião. I would also like to thank my co-adviser and professor at FCT UNL, Artur, for his valuable feedback and positive attitude towards the work developed.

”

“New ideas pass through three periods: It can’t be done. It probably can be done, but it’s not worth doing. I knew it was a good idea all along!”

— Arthur C. Clarke
(British Science Fiction Writer and Futurist)

Abstract

Real Estate is a cornerstone of the global economy, representing a significant portion of wealth and being a major factor in any country's GDP. Yet, despite being one of the most important sectors, it is still one of the most outdated. The traditional process of land conveyance is cumbersome, slow, and expensive, primarily due to the fragmentation of information across various controlling entities. There's a need to have this data unified in a single shared database that all the relevant parties may access and contribute, but none fully controls. This study explores the potential benefits of using blockchain technology for this purpose and proposes a novel smart contract architecture for a permissioned consortium blockchain network implemented with Hyperledger Besu. The presented solution comprises land conveyance and renting functionalities and demonstrates how its application could bring increased efficiency, transparency, and trustworthiness over legacy systems.

Keywords: Real Estate, Land Registry, Blockchain, Smart Contracts

Resumo

O setor imobiliário é um pilar da economia global, representando uma porção significativa da riqueza e sendo um fator crítico para o PIB de qualquer país. No entanto, apesar de ser um dos setores mais importantes, é também um dos mais desatualizados. O processo tradicional de compra e venda de imóveis é complicado, lento e de alto custo, principalmente devido à fragmentação de informação entre as várias entidades que controlam o processo. Há uma necessidade de unificar toda a informação numa base de dados única e partilhada que todas as partes relevantes possam aceder e contribuir, mas que nenhuma controle totalmente. Este estudo explora os potenciais benefícios do uso da tecnologia blockchain para esta finalidade, e propõe uma arquitetura de smart contracts para uma rede permissionada de consórcio, implementada com Hyperledger Besu. A solução apresentada compreende funcionalidades de transferência e aluguer de propriedades, e é demonstrado como a sua aplicação poderia trazer maior eficiência, transparência e confiabilidade em relação aos sistemas tradicionais.

Palavras-chave: Imobiliária, Registo Predial, Blockchain, Smart Contracts

Contents

List of Figures

List of Tables

Listings

Acronyms

BFT Byzantine Fault Tolerance *(p. 8)*
BTC Bitcoin *(pp. 8, 13)*

CA Certificate Authority *(p. 10)*
CFT Crash Fault Tolerance *(p. 8)*
CLI Command-line Interface *(p. 74)*
CNS Contract Name Service *(p. 43)*

Daml Digital Asset Modeling Language *(p. 17)*
DAO Decentralized Autonomous Organization *(p. 29)*
dApps Decentralized Applications *(p. 11)*
DMV Department of Motor Vehicles *(p. 8)*

EEA Enterprise Ethereum Alliance *(pp. 17, 37, 39)*
eIDAS European Identification and Trust Services Regulation *(p. 7)*
ERC Ethereum Request for Comment *(p. 12)*
ETH Ether *(pp. 8, 12, 13, 36)*
EU European Union *(p. 7)*
EVM Ethereum Virtual Machine *(pp. 11, 16)*

FCT NOVA School of Science and Technology *(p. 2)*

GDP Gross Domestic Product *(pp. 1, 26)*
GDPR General Data Protection Regulation *(pp. 15, 17, 85)*

IBFT Istanbul Byzantine Fault Tolerance *(pp. 16, 43, 78)*
IBPDI International Building Performance & Data Initiative *(p. 20)*
IPFS InterPlanetary File System *(pp. 20, 87)*

LLC Limited Liability Company *(pp. 23, 29)*

MiCA Markets in Crypto-Assets *(p. 14)*
MSP Membership Service Provider *(p. 24)*

NFTs Non-fungible Tokens *(pp. 13, 21)*

OCL Object Constraint Language *(p. 41)*
OSCRE Open Standards Consortium for Real Estate *(p. 20)*

P2P Peer-to-Peer *(p. 7)*
PKI Public Key Infrastructure *(pp. 10, 34, 37, 86)*
PoCs Proofs-of-Concept *(p. 25)*
PoS Proof-of-Stake *(p. 8)*
PoW Proof-of-Work *(p. 8)*

RESO Real Estate Standards Organization *(p. 20)*

SEC Securities and Exchange Commission *(p. 14)*
SPV Special Purpose Vehicle *(pp. 23, 29)*

TDD Test-Driven Development *(p. 80)*

UML Unified Modeling Language *(pp. 24, 41)*
URI Uniform Resource Identifier *(p. 20)*

1 Introduction

Real estate makes up a sizable portion of the world's economic assets - Statista's reports reveal that the value of the worldwide real estate market reached US$613.60tn in 2023 [62]. The land registry system is one of the crucial aspects of any government. Proper land record management is essential for economic growth and governance, as it constitutes a major factor in increasing a country's GDP. Even so, many countries rely on archaic, century-old procedures to manage their land records and conduct land transactions. These outdated methods are increasingly seen as nonsensical and are a source of widespread dissatisfaction. The most commonly spotted flaws are typically related to [58, 59]:

Time - The conveyance process may take weeks or even months to complete due to the large number of the required procedures and the need to physically meet with the involved stakeholders.

Complexity - The transacting parties are typically thrown at a labyrinth of procedures and bureaucracies that make the process confusing and difficult to go through without the assistance of an expert.

Cost - Many expenses and taxes are placed upon the transacting parties due to the involvement of numerous entities, significantly raising the transaction cost.

Fraud - Having different entities control different steps of the process results in a serious lack of transparency and creates opportunities for malicious individuals to exploit the system [29].

The root cause of these flaws lies in the decentralized and fragmented nature of the land conveyance process. Land records and property-related information are administered by a multitude of stakeholders, including government agencies, legal bodies, real estate agents, and financial institutions [27], each carrying their own roles and purposes. Data and documents necessary for the process completion are scattered across servers of more than one party, often duplicated and stored in different formats [30, 37]. The standardisation and transparency of information on real estate properties is a necessary step for the globalisation and increased efficiency of the real estate sector [33]. Ideally, all the data

should be unified in a single common database [17], which the various involved entities should be able to access and modify based on their role and permissions. However, it is important that none of the entities fully control the database since trust between them is not guaranteed. Blockchain technology may present a viable solution for this purpose. Blockchain is a decentralized, distributed ledger technology that offers transparency, security, and immutability. For this particular purpose, a consortium-typed permissioned blockchain network would be ideal, as it would enable the referred parties to access a shared database without the need for a central authority, while maintaining a level of privacy and confidentiality suitable for this sensitive data.

1.1 Objectives

The primary goal of this work is to study the feasibility of using blockchain technology as the core infrastructure to store land-related data and to demonstrate how its capabilities could be used to automate processes such as land conveyance, property leasing and mortgage financing. Throughout this study, we will examine current land record systems, highlighting their strengths as well as their limitations; provide an overview of previous studies and existing implementations, identifying the best practices and standards; and finally, based on the knowledge and insights gained, we will propose a novel smart contract architecture for a consortium blockchain network using Hyperledger Besu. This architecture aims to enhance land management systems' efficiency, reliability, and security, contributing to more streamlined and transparent property transactions.

1.2 Contributions

The expected contributions from this work are:

1. A report of the existing solutions for real estate property management;
2. The design, implementation, and evaluation of a smart contract architecture for a consortium blockchain network.
3. Network configurations relative to the roles and permissions of the peers participating in it;
4. A demonstrative client application to help showcase the system's capabilities.

1.3 Research Context

The Blockopoly Master Thesis proposal emerges from the collaboration between the academic expertise of NOVA School of Science and Technology (FCT) and the innovation initiatives of Crossjoin, a portuguese-based company created by a small group of experienced engineers mainly focused on the performance optimisation of large enterprise

systems. For the last 14 years and since the company's inception, Crossjoin has accumulated consulting expertise and knowledge in various industries, from Telecoms to Finance services and even going through Agriculture verticals. During its life, it increased the surface area of its activities, which now encompass specialised architecture consultancy and software engineering, application and infrastructure support, and overall performance optimisation programs. With a keen interest in the technology that will become mainstream in the following years, Crossjoin started cross-cutting initiatives for their clients with internal product development and technology exploration. These initiatives allow the company to experience the hardship of newly introduced technologies and work toward optimal insertion and maintenance paths. Crossjoin believes that its innovation efforts will yield a market advantage on consulting services for bleeding-edge technologies and corresponding performance optimisation since they've been through the problems their clients will face in the future.

This work stems from previous explorations with blockchain technologies, resulting in a corporate mobile application portal for partnership benefit claims such as shared expenses or loyalty programs. Crossjoin understood from this R&D effort that the organisation could start initiatives integrating shared client needs. This thesis proposal is part of a more comprehensive research program for an integrated platform for private and public asset management and transactions, which connects requirements and needs from one governmental client with a few private ones. The company's objective is to showcase the outcome of this specific initiative in tandem with other ongoing developments.

1.4 Document Structure

The rest of the document is structured as follows: In chapter 2 we provide an overview on diverse topics that are pertinent to the project and discuss relevant technologies that were considered before development. Next, chapter 3 presents approaches and frameworks proposed by other authors, as well as government initiatives and solutions already available in the market that are relevant to the work. In chapter 4, we focus on defining the scope of the project and give insights on the main components of the architecture. In chapter 5 we finally reveal the proposed implementation, explaining the design choices behind it, and lastly in chapter 6 we present the evaluation methodology utilized to validate the solution.

2

Background

2.1 Current State of Land Conveyancing

Real estate conveyance is a process in which several intermediaries and authorities might be involved, and where different procedures and bureaucracies might take place depending on each country's legislation. This work will not target the system of any specific country. Instead, we will try to generalize the land conveyance procedure and take only into consideration the characteristics that are present in most of them. To achieve this, it is essential to firstly dive into a simplified view of the general flow of the process. The following section provides a summarized overview of the works done by Garcia-Teruel [27] and Shuaib et al. [57, 58].

The first step in a land deal is usually for the parties to hire **real estate agents** - professionals that help them find, buy, or sell a property and aid them during the transaction process. They provide guidance on market trends, property valuation, and legal requirements. Then, an entity must responsible for tasks such as verifying the identity of the transacting parties, preventing illegal activities and ensuring the legality and effectiveness of land transactions. In Western European countries, that entity is the **notary**. In Nordic countries, the whole process is completed by real estate agents or by **lawyers**. England and Ireland have a system of **solicitors** or **licensed conveyancers** for this purpose. If we add mortgage loans to the mix, **Banks** are also typically involved, along with lawyers and **property valuators**.

Pre-agreement contracts are often drafted before the final transaction takes place. These documents serve as a formal declaration of intent and ensure both parties are committed to the transaction. They outline the basic terms and conditions of the sale, such as the agreed price and payment schedule, but may also include several **contingency clauses** that refer to conditions that must be met before the sale agreement becomes binding. These can relate to almost any need or concern a party may have regarding the property's appraisal, financing or inspection. During the following days, known as the contingency period, these clauses are verified through property appraisal and inspection. If any contingencies

are not met, the affected party gets the right to renegotiate or to back out of the deal without legal consequences. On the other hand, if every contingency clause is met, the contract becomes legally enforceable. If a party were to back out at this stage, it would be in breach of contract, meaning it could face legal consequences. During this time, the buyer must also secure **mortgage financing**, if applicable.
Finally, several documents may be required depending on jurisdiction. Examples include title deeds, bank statements, insurance proofs, energy certificates, government-issued permits, licenses relevant to the property, etc. These documents are often issued by various other entities and valuators. When all the requirements are finally met, both parties sign the **sale deed**, the legal document confirming the transfer of ownership from the seller to the buyer. It serves as the definitive proof of the transaction. Once the transaction has been completed, it shall be registered in the land registry by the competent authority, which could be a governmental authority, the courts, or an independent public authority. Table 2.1 describes a generic sequential flow of how a real estate transaction is usually processed, although exceptions may apply depending on each case [17, 58].

	Buyer	**Seller**
1.	Initiation through engagement of a real estate agent or lawyer	
2.	Mortgage loan due diligence (if applicable)	Sale legality verification and registration procedures
3.	Property visitation and negotiation phase	
4.	Drafting and signing of the letter of intent (pre-agreement contract)	
5.	Contingency Period Securing mortgage financing Gathering of necessary documentation	Contingency Period Gathering of necessary documentation
6.	Finalization of the sale agreement and execution of the financial transaction	
7.	Official recording of the transaction in the Land Registry by the designated authority	
8.	Handover of the keys	

Table 2.1: Steps in Land Conveyance

2.2 Property Leasing

Property leasing, on the other hand, is generally a simpler and less formal process. It usually does not require the participation of any intermediaries and is not even required to be registered in the land registry, although it shall be declared to authorities for taxation purposes. This lack of registration, however, creates opportunities for underreporting of rental income leading to tax evasion.
Landlords and tenants can draft various types of leases, each with its own set of terms and conditions. The **rental** or **lease agreements** outline the rights and responsibilities of both parties during the lease term. Common clauses found in rental agreements include details regarding the rental amount, payment schedule, duration of the lease, security deposit requirements, maintenance responsibilities, and terms for lease termination or renewal. Additionally, leases often specify rules regarding property use, pet policies, subleasing arrangements, and penalties for breaching the terms of the agreement.
In the event that either party breaches the terms of the lease agreement, certain actions may be taken. If the tenant fails to pay rent or violates other terms of the contract, the landlord may initiate eviction proceedings or pursue legal action to recover damages. Conversely, if the landlord breaches any obligations, the tenant may have the right to prematurely terminate the lease or to seek monetary compensation.

2.3 Digital Signatures

In broad terms, a signature is a mark that allows us to know that something has been endorsed by a certain individual or entity, as it is unique and distinctive, so only the signer can replicate it.
In traditional paper-based agreements, handwritten signatures hold significant legal weight, as they symbolize the party's agreement to the terms of the contract. However, this is neither the only way to sign agreements nor is it the most secure. Digital signatures, also called electronic signatures, have been widely recognized and adopted for decades, with their development tied to advancements in cryptography. At their very core, they consist of two unique cryptographic keys, a public key and a private key, that work together through mathematical algorithms that can be used to validate any digital message's authenticity and integrity. Only the owner of the private key can reproduce signatures, hence their uniqueness.
Nowadays, digital signatures are utilized across various applications and tools. They form the bedrock of Internet security, employed in everything from online banking applications to email correspondence. When signing contracts and agreements, a qualified digital signature carries the same legal weight as a handwritten one. There's no longer the need to sign documents manually in modern days. Electronic signatures offer speed, cost-effectiveness, and enhanced security, as they cannot be forged as easily as manual ones often are.

Digital signatures create a virtual fingerprint unique to a person or entity, and thus they can also be used as a nationwide identification method. Many countries have implemented national civil identification systems based on cryptographic key pairs. For instance, Portugal has its "Chave Móvel Digital" (Digital Mobile Key), while other countries have similar systems. The standardization efforts of regulations like European Identification and Trust Services Regulation (eIDAS) aim to harmonize these systems across the European Union (EU), facilitating seamless and legally binding cross-border transactions, contributing to a more integrated digital economy.

2.4 Blockchain Overview

The term "blockchain" was initially popularized following the publication of the Bitcoin whitepaper in 2008 [43], by an anonymous author under the pseudonym of Satoshi Nakamoto. Although the term itself is not mentioned in the whitepaper, it began to be widely used to describe the underlying technology of Bitcoin. Since then, many other solutions have emerged utilizing similar or derived technologies, following the core principles of Bitcoin. Years of innovation and refinement have broadened the term's applicability, making it far more versatile than its original counterpart.
Blockchain is an innovative database technology that can be described as a digital and distributed ledger of transactions. These transactions are digitally signed and recorded in the form of blocks securely linked together via cryptographic hashes, effectively forming a chain, hence the name blockchain. Instead of relying on a centralized location managed by a single entity, blockchains are typically managed by a distributed Peer-to-Peer (P2P) computer network composed of several nodes that can belong to different authorities who may not trust each other. This decentralized nature along with the following characteristics are what make blockchain so different from other database solutions [69]:

Consensus - For a new transaction to be recorded, most of the nodes in the network must agree on its validity and inclusion.

Transparency - Transactions recorded on a blockchain are visible to all network participants, making it possible for anyone to verify the integrity of the ledger.

Immutability - Blockchain transactions are irreversible. Once a block has been recorded, all transactions it contains are executed and cannot be undone. In some cases, however, they can be rollbacked.

Tamper-Resistance - Each block in the chain contains a cryptographic hash reference to the previous block, meaning that if a block were to be tampered with, all subsequent blocks would also need to be recalculated. Due to the consensus algorithmic security, this is an extremely difficult task.

Blockchains as we know today have grown to be much more than just an innovative way to store signed transactions. In some blockchains like Ethereum, any computable function may be implemented with Solidity, its Turing complete programming language for writing business logic. This allows blockchains not only to function as a way to store data and execute transactions but also as a way to autonomously enforce contract terms or even implement an entire system's logic. These features have led many industries to explore blockchain's opportunities in business areas, including Finance, Energy, Mobility, Smart Cities, IoT systems, and more. As a recent example, California's Department of Motor Vehicles (DMV) digitized 42 million car titles on the Avalanche blockchain in a bid to detect fraud and smoothen the title transfer process, according to Reuters [60]. The project was the fruit of a collaboration with Oxhead Alpha, and plans were underway every since 2020 [15].

2.4.1 Public, Private and Consortium Blockchains

Blockchain ecosystems can be categorized based on network access as public, private or consortium [69]. Public Blockchains are usually open and permissionless, meaning anyone can set up a node and become part of the network by contributing to the ledger. This means that the blockchain's content is fully visible to the public and that every peer is equal and has the same set of roles and permissions. This is the most disruptive and innovative blockchain type and, therefore, the most well-known. Bitcoin and Ethereum are the most popular examples. Public Blockchains are also closely associated with cryptocurrencies since there is a need to incentivize people to participate in the network. Each of these blockchains may have a native cryptocurrency token (e.g. Ether (ETH) for Ethereum, and Bitcoin (BTC) for Bitcoin) and by contributing to the ledger and behaving correctly, nodes are rewarded with a certain amount of that cryptocurrency. The way these tokens are generated involves the use of the consensus algorithm. The two most common public blockchain consensus algorithms are the Proof-of-Work (PoW) and Proof-of-Stake (PoS), although there are many more.

In contrast, Private Blockchains are fundamentally different in their structure and purpose. These blockchains are closed and permissioned, meaning access to participate in the network is restricted to a pre-selected group of entities. This controlled access ensures that on-chain data remains within the trusted group, making private blockchains ideal for industries that deal with confidential data, like finance, healthcare, and real estate. Because in these networks the number of nodes is small compared to those of public blockchains, the consensus mechanisms are also quite different. Since there is no need to incentivize random people to participate in the network, there is no need to have a native cryptocurrency system. Without cryptocurrency mining, private blockchains may use more efficient consensus algorithms like Byzantine Fault Tolerance (BFT) or Crash Fault Tolerance (CFT). These algorithms are designed to be efficient and reliable in smaller

networks where the peers are known and trusted, resulting in faster transaction speeds and much lower processing costs than public blockchains.

Permissioned networks that are managed by various organizations, instead of by only one, are commonly referred to as Consortium Blockchains [68]. This model is beneficial when multiple organizations must collaborate and share data securely without fully trusting each other. Consortium blockchains typically provide ways to define roles and permissions for each of the network's peers, thus facilitating controlled access and activity based on predefined rules and criteria. For this reason, consortium blockchains often provide a middle ground between public and private blockchains, combining the transparency and collaborative aspects of public blockchains with the enhanced security and control found in private ones. Table 2.2 was originally proposed by Dib et al. [18] and highlights the technical differences between each of the discussed blockchain types.

Property	Blockchain Governance		
	Public	Consortium	Private
Governance Type	Consensus is public	Consensus by a set of participants	Consensus by a single owner
Transactions Validation	Anynode (or miner)	A list of authorized nodes (or validators)	
Consensus Algorithm	Without permission (PoW, PoS, PoET, etc.)	With permission (PBFT, Tendermint, PoA, etc.)	
Transactions Reading	Any node	Any node (without permission) or A list of predefined nodes (with permission)	
Data Immutability	Yes, rollback is almost impossible	Yes, but blockchain rollback is possible	
Transactions Throughput	Low (few dozen per second)	High (hundreds/thousands transactions per second)	
Network scalability	High	Low to medium (a few dozen/hundred of nodes)	
Infrastructure	Highly-Decentralized	Decentralized	Distributed
Features	Censorship resistance Unregulated and cross-borders Support of native assets Anonymous identities Scalable network architecture	Applicable to highly regulated business Efficient transactions throughput Transactions without fees Infrastructure rules are easier to manage Better protection against external disturbances	
Technologies	Bitcoin, Ethereum, Ripple, etc.	MultiChain, Quorum, HyperLedger, etc.	

Table 2.2: Blockchain Classification. Source: Adapted from [18]

2.4.2 Public Key Infrastructure (PKI)

Beyond the peer nodes, responsible for maintaining the blockchain's ledger and validating transactions, other participants interact with the blockchain, such as end users and client applications. These are the individuals who use the system for its intended purposes, signing and submitting transactions to be validated by the nodes.
Each user is uniquely identified by a cryptographic key pair, consisting of a public and a private key. These key pairs are the foundation of blockchain's security guarantees, ensuring transaction authenticity and integrity. As the name suggests, the public key is made public and may be openly shared. It may be used by others in the network to encrypt messages that only the holder of the corresponding private key can decrypt. Within the context of a blockchain network, it primarily serves as a user's identifier or address. The private key, on the other hand, is secret and should only be known by the user, as it

serves to digitally sign transactions in its name. Others may validate signatures using the corresponding public key. Public and private keys are intrinsically linked as the public key is generated by the private key itself. However, due to the cryptographic nature of the key pair, it is not feasible to derive the private key from the public key [61]. Finally, each public key is authenticated and verifiable through a public key certificate, a digital certification provided and issued by a Certificate Authority (CA), which confirms the legitimacy and trustworthiness of the public key. X.509 is the standard format for public key certificates, widely used in various secure internet protocols. Public Key Infrastructure (PKI) is commonly defined as the set of hardware, software, people, policies, and procedures needed to create, manage, store, distribute, and revoke digital certificates.

In public blockchains, the management of the key pairs is at the responsibility of the individual users. The network itself provides PKI support, enabling signature verification across different independent nodes. However, the public key is the only known information about the user, meaning that each account is intrinsically anonym. Because of this, it is very complex to establish a one-to-one relationship of key pairs to real-world identities, as a user may create many key pairs. In some cases where possible, these connections are typically made through processes outside, and not explicitly supported by the blockchain itself [68].
In contrast, permissioned blockchains have a more centralized approach to PKI, with designated authorities responsible for issuing and managing the key pairs. This centralized approach makes private networks best suited for scenarios where the generation of key pairs must be regulated and makes it easier to connect them to real-world entities. They can offer other advantages, such as facilitating user authentication through conventional methods like usernames and passwords, and providing additional recovery options in case of lost credentials. If a permissioned blockchain system were to be implemented nationwide, it could potentially leverage existing digital signature systems used by countries that support such infrastructures. This integration would allow the blockchain to utilize national digital key pairs to authenticate citizenship within the system, tracing each transaction to a real-world identity.

2.4.3 Smart Contracts

In 1994, Nick Szabo first defined a smart contract as a "computerized transaction protocol that executes the terms of a contract" [63], being a contract an agreement that can be enforced by law. While the core concept remains the same, the smart contracts that we know today refer to code deployed in blockchain networks that comprise any programming logic and may perform transactions autonomously. They are equivalent to application code but with a decentralized nature. Instead of being executed in a central location, they are executed concurrently on the nodes of a blockchain network, creating transactions that modify the network's global state. Thus, smart contracts are responsible for defining

the rules for the transactions that occur on the network.
Not every blockchain infrastructure supports smart contract deployment, and for the ones that do, the programming language used to define their logic may differ. Ethereum, the pioneer of smart contracts, uses Solidity for their specification. In private blockchain development, it is common for frameworks to allow smart contract implementation using general-purpose programming languages such as Python, Java, and Javascript. According to the blockchain properties previously discussed, smart contracts are immutable once deployed, meaning the contract cannot be changed or broken, ensuring a high level of security and trustworthiness. They ultimately act as transaction escrows, removing the need for additional intermediaries.

2.5 Ethereum

Ethereum is a revolutionary blockchain technology that was first introduced in 2015 by Vitalik Buterin and his team [9]. Unlike Bitcoin, which is primarily a digital currency system, Ethereum brings a platform where developers can create diverse Decentralized Applications (dApps) that do not rely on traditional financial institutions or intermediaries, through the introduction of smart contracts.
The Ethereum Virtual Machine (EVM) is at the core of its functionality. The EVM is a powerful, Turing-complete virtual machine run by every network node in order to calculate the result of smart contract execution and maintain consensus across the blockchain. Thus, smart contract execution happens directly within the network.
Solidity, Ethereum's programming language, is designed specifically to create and implement smart contracts. It provides an interface similar to general-purpose programming languages but introduces functionalities that allow developers to write logical operations within the context of a transaction.
Ethereum was developed with the vision of establishing a broad, public blockchain network known as the Ethereum Mainnet. However, in addition to this main network, there are other public branches maintained by different participants, test networks, and significant utilization of this technology in enterprise private environments. For this latter purpose, specific frameworks extend the core functionalities of Ethereum to enhance its applicability in permissioned contexts. Examples of such frameworks include Consensys' Quorum [14] and Hyperledger Besu [32].

2.5.1 Gas

In Ethereum, gas is a fee required to conduct a transaction on the network successfully. It functions as a unit of measurement for the amount of computational work required to execute operations. The exact price of the gas at the time of a transaction is determined by the network capacity and congestion.
Gas is a required in public blockchain networks since it acts as a mechanism to prevent spam

and inefficient network resource allocation. Users pay for the computational effort with tiny fractions of ETH, the native cryptocurrency, which prevents malicious individuals from overloading the network by deploying resource-intensive code as it comes at a significant monetary cost.
In permissioned networks, the use of gas is less common. This is because the number of nodes is largely reduced compared to public blockchains. Thus, the total computing power spent on processing each transaction is also very inferior. Many enterprise blockchain frameworks do not implement a transaction fee system since they typically do not support a native cryptocurrency. Additionally, as a result of being private, it is easier to detect malicious individuals and revoke their access to the network.

2.5.2 ERC - Ethereum Request for Comment

Ethereum Request for Comment (ERC) is a protocol within the Ethereum ecosystem for proposing improvement requests to the Ethereum network. After careful review from the community, if the request is accepted and implemented, it becomes a standard, retaining the original ERC identifier as the standard name. ERCs are most famously used for creating token standards, which define common rules that all compliant tokens on the Ethereum network must follow. This standardization ensures that tokens will operate predictably across various applications and wallets within the Ethereum ecosystem.
Nevertheless, adapting these standards to private solutions may still be valuable. They offer a minimal and yet rich list of secure and versatile endpoints, which should allow for the execution of nearly any functionality that might be desired within the system. By adhering to these standards, solutions benefit from enhanced modularity, scalability, and independence from other system components. Furthermore, in the context of this thesis, following these standards not only brings credibility to the solution but also promotes the development of a generic architecture that can serve as a blueprint for future work. This approach aligns with the ideal scenario for thesis outcomes, setting a foundation for further research and development within the blockchain technology landscape.

2.6 Oracles

An intrinsic limitation of smart contracts is their inability to inherently interact with data or systems outside their native blockchain environment. Without the ability to interact with the external world, the potential use cases of blockchains would be highly restricted. Oracles address this limitation. In the context of blockchain technology, an oracle is an entity that acts as a mediator between blockchain environments and external systems. They play a foundational role in the creation of the verifiable web, enabling smart contracts to receive inputs and outputs from the real world. There are many types of oracles, each serving distinct purposes. Examples include real-time asset price tracking, weather monitoring, and verifying the outcomes of real-world events.

An oracle can be software or hardware-based, capable of executing code stored on-chain and possibly equipped with physical sensors or the ability to manipulate real-world devices [38]. But by definition, oracles can also be human, providing necessary information manually when automated systems cannot suffice.
Oracle services are usually tied to some centralization. Still, it is possible to leverage decentralized oracle networks, which offer the advantage of providing data verified by multiple separate oracle entities instead of relying on a single data source. This decentralization enhances the reliability and trustworthiness of the data, mitigating risks associated with a single point of failure or data manipulation.

2.7 Tokenization

One of smart contracts' most important capabilities is their ability to create and manage digital assets, which we call blockchain tokens. Tokens are more of a public blockchain concept, as most permissioned blockchain frameworks don't directly support token standards or libraries. However, there may be times when it can make sense to replicate their logic in private networks, depending on the system requirements. Tokens have an owner, but their ownership might change. Based on their interchangeability, they can be classified into fungible or non-fungible [26].

Fungible Tokens refer to tokens whose units are equal and interchangeable with each other since they hold the same value and properties. This means that one unit of a fungible token can be exchanged on a one-to-one basis with any other unit of the same token type, as they have equal value. Ownership over this kind of token can be seen as "how many units of X token does user Y have?". They may also be divisible into smaller units. By this definition, cryptocurrencies themselves, like BTC and ETH, are fungible tokens. However, they can be used for a variety of other things, such as representing points in a ranking system, voting rights, or even shares over an asset. In Ethereum, ERC-20 is the most popular standard for fungible token definition.

Non-fungible Tokens (NFTs), on the other hand, are unique and cannot be exchanged on a one-to-one basis with each other, although they can still be traded and exchanged for money, cryptocurrencies, or other NFTs. Each NFT is indivisible and has its own value and properties. They are used to represent ownership and proof of authenticity of a unique digital item, such as digital art and collectibles. These tokens are created through a process called minting, usually initiated through a mint function defined in the NFT smart contract, in which the information such as its owner and its metadata are provided. After this information is validated by the network, a block comprising the NFT is submitted on the blockchain, and the NFT is minted. Ethereum was the first blockchain introducing NFTs, being ERC-721 the most common standard for non-fungible token

definition, although many other notable blockchains have also adopted the concept.

Although tokens are fundamentally digital assets, they can act as a digital representation of a real-world physical asset. "Tokenization" is the process of converting rights of an asset into a digital token on a blockchain network. To these tokens we call Security Tokens [16]. There are multiple advantages that justify why someone would want to tokenize an asset. One of them is to increase the liquidity of assets that are traditionally illiquid or hard to transfer, such as real estate or company shares. By creating digital tokens, these assets can be easily traded on digital platforms since systems can be developed to simplify the exchange process and remove intermediaries. Another advantage is the ability to divide the asset into smaller parts, allowing for fractional ownership. Finally, by keeping the asset ownership recorded on a blockchain, it becomes possible to view the complete history of ownership, enhancing transparency and trust in the transaction process. Tokenizing an asset is not a trivial process, however. Security tokens must often comply with various regulatory frameworks depending on the jurisdiction. The U.S. treats security tokens similarly to traditional securities, applying existing securities laws through the Securities and Exchange Commission (SEC). The EU has been working on creating a comprehensive framework for digital assets, including security tokens, through the Markets in Crypto-Assets (MiCA) regulation [24], which entered into force in June 2023. This is a very hot topic with a lot of research going on in recent years, which reflects the growing importance of digital assets in the financial industry. The ERC-1400 was the first and until very recently the only Ethereum standard for security tokens. In December 2023, T-REX Protocol [64] achieved 'final' status and was recognized as the ERC-3643 security token standard. These standards use different approaches, but both of them focus on enabling the enforcement of compliance rules and the control of transfers to eligible investors.

2.8 Data Privacy and Protection

Any system that stores data related to individuals is subject to regulations and obligations imposed by the local legislation under which it operates. If the goal of this study is to explore the feasibility of using blockchain in the real estate industry, then it is essential to examine how blockchain can meet these requirements. In fact, it is here that some paradoxes concerning this technology begin to emerge.
Building a system of this kind inevitably involves storing various kinds of data. Firstly, there's data related to the properties themselves. This includes property metadata (information regarding the property's physical description, ownership, location, valuation, and transaction history) and also property-related documentation, which may be issued by different stakeholders. This information is less sensitive, as most of this data should be public and is in most countries. This is because real estate is recognized as an economic

activity or sector, placing real estate properties as products on the market for the transmission of use or ownership. Their information must be accessible to anyone interested in knowing the situation and features of the property [33]. Often, the challenge is that the information is public but difficult to access. However, although property data is intended to be public, the processes involving them also require parties to provide personal data, mainly of a financial nature, which is considered sensitive. The issue with personal data is that it is often protected by regulations and laws that define what kind of data can be collected and give individuals the right to choose how their data will be used. Within European boundaries, for example, organizations must comply with the General Data Protection Regulation (GDPR) [66], which is considered to be one of the strictest data protection regulations globally.

One of the primary conflicts inherent in blockchain technology arises from its immutable nature, conflicting directly with GDPR mandates such as the Right to be Forgotten (Article 17) and the right to rectify inaccuracies in personal data (Article 16) [12]. The immutable characteristic of blockchain means once data is entered, it cannot be altered or erased, thus creating a paradox when applied to environments requiring compliance with such privacy laws. Furthermore, identifying roles such as data controllers and processors becomes problematic. Since the data might be public, it is often unclear how it is used or for what purposes. This ambiguity complicates compliance with GDPR requirements such as the right of access, where data subjects must be able to inquire how their data is being processed. In fully decentralized applications, users of the network might all be considered data controllers, thereby complicating the enforcement of data subject rights [35]. Cross-border data transfers are another significant area of concern under GDPR. Public blockchain's global, decentralized nature means it can be impractical to ensure all nodes comply with specific data protection standards.
In permissioned networks, however, these challenges are somewhat mitigated as there is control over the network's participants. This makes it possible to enforce consistent data protection standards across all participants, aligning more closely with GDPR requirements [35]. Enterprise blockchain technologies may also leverage privacy mechanisms that are not present in public environments, as we will discuss further ahead, in section 4.4. Furthermore, depending on the blockchain technology, it may be possible to amend or even erase data entirely from the ledger, by resorting to a consensus among the consortium members. If this is not feasible, alternatives might include storing sensitive data off-chain or employing data anonymization techniques, thereby rendering the data exempt from GDPR.
All this is to say that there are ways around these issues. The biggest challenge is to find a solution that responds to all of them simultaneously.

2.9 Enterprise Blockchain Frameworks

The blockchain framework is the tool that will allow us to deploy the blockchain network and code the business logic, defining the laws within the system. In this section, we present an overview of some of the different frameworks available for developing permissioned enterprise blockchain networks.

2.9.1 R3 Corda

Corda [8] is an open-source permissioned blockchain platform developed by R3. It is particularly suited for finance-related applications as it has a strong focus on facilitating legal processes. One of its great features is its enhanced interoperability, as the platform is able to seamlessly interact and integrate with various systems and networks. CorDapps, Corda's solution for implementing business logic, supports development both in Kotlin and Java. They are significantly different from smart contracts as they are more structured and designed to be legally enforceable.

2.9.2 Hyperledger Fabric

Hyperledger Fabric [31] is an open-source permissioned distributed ledger technology hosted by the Linux Foundation. It is best known for its high flexibility and modularity, as it allows network designers to seamlessly plug in their preferred components like consensus and membership services. Fabric follows a contract-like architecture for implementing the business logic, commonly referred to as chaincode. They offer support for developing chaincode in various programming languages such as Go and Javascript.

2.9.3 Algorand

Algorand [11] is primarily recognized as a public, decentralized blockchain network, but it also offers capabilities for deploying permissioned networks. The code for Algorand is open-source so it can be cloned and used in private blockchains. Algorand supports the creation and management of tokens and smart contracts, and utilizes a unique consensus mechanism called Pure Proof of Stake (PPoS), which enables the network to handle high transaction throughput with minimal fees.

2.9.4 ConsenSys Quorum

Quorum [14] is a permissioned blockchain framework that enables enterprises to leverage Ethereum for their private blockchain applications. It utilizes a modified version of Ethereum's protocol, keeping the core technology such as its runtime environment (EVM) and solidity smart contracts, but offering private consensus algorithms like IBFT, Raft and other permissioned network capabilities. This allows solutions to benefit from the robust

development tools and testing environments that Ethereum offers while keeping them completely independent from Ethereum's public network.

2.9.5 Hyperledger Besu

Similar to Quorum, Hyperledger Besu is also an enterprise Ethereum client. In fact, both conform to the Enterprise Ethereum Alliance (EEA) specifications, which standardize protocols to ensure interoperability and common practices among enterprise Ethereum applications. While they share many core features, such as the private consensus algorithms and network permissioning mechanisms, Besu leverages enhanced compatibility with the broader Ethereum ecosystem and provides richer functionalities in terms of data privacy.

2.9.6 Digital Asset's Daml and Canton

Canton is a ledger interoperability protocol that connects multiple Daml ledgers belonging to different domains into a single virtual global ledger [19]. Digital Asset Modeling Language (Daml) is a smart contract language specifically designed for complex multi-party agreements, bringing incomparable capabilities to ensure privacy and compliance over data. Daml essentially allows us to strictly define who is entitled to see and sign any kind of information within the network. Canton blockchains enforce these visibility and authorization rules and ensure transaction integrity with very high levels of privacy. In fact, the main selling point of Daml and Canton is its privacy control mechanisms and the possibility of complying with GDPR. Because Canton is a relatively recent tool, many of its appealing features are still in the early stages of development. Such is the case for broad interoperability features, such as integrating other blockchain ecosystems like Hyperledger Fabric and Besu networks, and even centralized infrastructure like Postgres databases.

2.10 Framework Comparison

We now present a side-to-side comparison of the previously presented frameworks, regarding the following aspects that we found relevant to the scope of the project:

Programming Language - The programming language used to code the logic differs from framework to framework. This choice can impact the ease of development, available libraries, and integration with other systems.

Logic Implementation - Not all of the presented frameworks use the concept of smart contracts. In fact, there's not much reason for a permissioned blockchain framework to follow the traditional smart contract model. R3 Corda, for instance, uses CorDapps that function differently from traditional smart contract architecture.

Native Token Support - The concept of tokens is generally more associated with public blockchains, mainly Ethereum. Some frameworks, however, provide ways of natively implementing tokens similar to those of Ethereum, using libraries or built-in functionalities.

Access Permissioning - One of the most important aspects when designing a consortium blockchain is defining peers' and participants' roles and permissions. This can be done natively, with integrated mechanisms within the technology (such as those in Hyperledger Fabric), or at the level of smart contracts if the tool supports such integration.

Privacy - Privacy is one of the most important aspects due to the sensitive nature of the data involved in a sector like real estate. It is essential that the chosen framework provides ways to ensure customized privacy for various types of data.

Time-based Event Support - If the framework provides ways to implement smart contracts or functions that autonomously trigger based on time events. This could be useful for implementing recurring transactions, such as rent payments triggered each month.

Development Tools - Robust development tools and comprehensive testing mechanisms provided by the framework can greatly facilitate the development and evaluation process.

Complexity - Given that the solution is a Proof-of-Concept and the time frame is limited, the overall complexity and learning curve of each framework represent significant factors to consider. Simpler and more intuitive frameworks may be more suitable under these constraints.

Table 2.3 summarizes the considered characteristics of each of the frameworks analyzed.

Framework	**Hyperledger Fabric**	**R3 Corda**	**Ethereum-Based (EEA)**	**Canton**	**Algorand**
Progr. Language	Golang	Java, Kotlin	Solidity	Daml	Python
Logic Implementation	Chaincode	CorDapps	Ethereum SC's	Daml Contracts	ASC1
Native Token Support	No	Yes (SDKs)	Yes (ERC)	Yes	Yes (ASA)
Access Permissioning	Native (RBAC)	Native (RBAC)	Smart Contract-based	Daml (Unmatched)	-
Privacy	Channels	Private Transactions	Private Transactions	Daml (Unmatched)	Co-Chains
Time-based Event Support	No	Yes	No	No	No
Development Tools	Robust	Robust	Comprehensive	Weak	Solid
Complexity	Complex	Moderate	Simple	Moderate	Moderate

Table 2.3: Comparison of Permissioned Blockchain Frameworks

3

Related Work

A very valid question that may come up is "Why Blockchain? Can't we use any other technology we already know?" Blockchain technology may neither be the only nor the best solution to this problem. But there are many reasons why it looks like one of the best contenders. As previously explained, blockchain can provide a shared database and ways to deploy code and build entire applications. This can be used to automate the processes of land conveyance, property leasing, and mortgage loan financing by autonomously enforcing agreement terms through smart contracts. But the options don't end here. In a utopian scenario, all operations related to real estate could be executed over a blockchain infrastructure. Many of these processes that are originally manual and require efforts from every stakeholder involved could be fully automated and done remotely through smart contracts and digital signatures. The smart contracts would execute the business flow without the opportunity to be manipulated, thus reducing corruption and fraud. Furthermore, blockchain may bring opportunities that cannot be achieved in the current infrastructure. It would maintain a complete and transparent history of details regarding every deal made concerning each property, improving the search process and helping in the decision-making for buying and renting real estate. By removing intermediaries and unnecessary steps, it could greatly decrease the overall cost of the process. Moreover, it would bring new and improved ways to invest in the real estate market, as tokenizing real estate assets highly increases liquidity and allows fractional ownership. Some of these opportunities are discussed in greater detail in the following sections, as we present related work done by other authors, companies, and governments.

3.1 Property-Related Data & Documentation

The first relevant point of discussion concerns the storage methodology of the actual data within the system. This includes property information, ownership records, financial data, market data, legal and regulatory information, transaction history, and also any kind of documentation and certificates that are required to execute most operations over that

property [30]. In a real-world scenario, tokenizing a property would require standardizing property information into a universal format shared by all stakeholders. This task by itself is extremely complex, as there is no universally recognized property classification methodology. Each country has its unique definitions and standards which are, in turn, products of its historical evolution [33]. On top of that, within the same country, it is often that different real estate agencies adopt varying formats and keep sale information and real transaction prices undisclosed [37], leading to a market that is opaque and driven by speculative pricing. The standardization of real estate property information is a necessary step for the globalization and increased efficiency of the real estate sector, but deciding on a global standard for property tokenization is beyond the scope of this work. There are already several studies on the key elements that need to be considered to create standard conventions for the description and valuation of properties [30, 33, 37] and significant efforts have been made by international initiatives like the Open Standards Consortium for Real Estate (OSCRE), the International Building Performance & Data Initiative (IBPDI), and the Real Estate Standards Organization (RESO) on developing comprehensive data models and standard proposals.

Property-related documentation suffers from a similar issue. Each country has its specific requirements, and depending on the jurisdiction, different documents are necessary to conduct sales and ensure that a property is compliant with local regulations. Even so, it is within the interest of this work to explore how blockchain technology can be utilized to issue, store, transfer, and validate property documents, potentially bringing about compliance improvements in property sales and rentals.

Consensys' ERC-1400 umbrella for security tokens encapsulates a suite of smart contract standards that aim to streamline on-chain security representation and management. Since security tokens are backed by some kind of physical asset (the security), they are often subject to regulatory requirements and therefore necessitate mechanisms to manage compliance-related documentation. Thus, ERC-1400 contracts offer features that allow the storage of document hashes to ensure integrity. However, the documents themselves are not stored on-chain, which is logical since storing an entire document on a public blockchain would be impractical due to the high space requirements, making it prohibitively expensive. Instead, a URI is provided so clients may download the document from an external source, like an IPFS network, and then use the hash to ensure the file was not tampered with.

```
interface IERC1643 {
    function getDocument(string memory _name) external view returns (string
        memory, bytes32, uint256);
    function setDocument(string memory _name, string memory _uri, bytes32
        _documentHash) external;
    function removeDocument(string memory _name) external;
    function getAllDocuments() external view returns (string[] memory);
    event DocumentRemoved(string indexed _name, string _uri, bytes32
        _documentHash);
    event DocumentUpdated(string indexed _name, string _uri, bytes32
        _documentHash);
}
```

Figure 3.1: ERC-1643 Document Management Standard Interface

This approach makes complete sense in environments where the blockchain acts as a supporting infrastructure with the goal of ensuring integrity over some process. This is the case for most real-life applications, as we are still far from a fully decentralized future. Digitizing and issuing documents on the blockchain would necessitate a substantial transformation and efforts from both the government and the various entities that currently manage and issue these types of records. This shift involves not only technological adoption but also significant changes in legal frameworks and data handling protocols. Therefore, most blockchain solutions seek to deliver the benefits of smart contracts while minimizing risk and resistance by ensuring minimal disruption to traditional systems.

3.2 Fractional Ownership of Assets

A topic that is frequently discussed among many authors is how to represent fractional ownership of assets in blockchain platforms. As previously explained, in public blockchains, assets such as real estate properties are frequently represented through Non-fungible Tokens (NFTs). However, a challenge arises as NFTs only support single ownership, meaning a single token can only belong to a single owner. There are a few ways to circumvent this. One method involves using fungible tokens such as ERC-20 tokens to represent ownership shares in an asset. For instance, a single real estate property can be represented by a fixed number of ERC-20 tokens, each representing an equal share of the property. In the same way, Gupta et al. [29] considered using ERC-777, another standard for fungible tokens with added features. A hybrid model can also be considered, combining features of both NFTs and fungible tokens. The real estate asset may be represented through an NFT for uniqueness, and then linking this NFT to fungible tokens that represent fractional ownership shares. Joshi et al. [34] suggested implementing this approach by using the ERC-1155 multi-token standard for real estate tokenization. Unlike the previous standards, the ERC-1155 can mint both fungible and non-fungible tokens, eliminating the complexity of handling two token standards and their interaction. In the proposed model, each

property is associated with a unique NFT and fungible token type within the ERC-1155 contract. Owning a share of the fungible tokens minted for a property equals owning a fraction of that property.

3.3 Rental Opportunities

The real estate rental market is another sector in need of innovation and improvement, afflicted by inefficiencies and a significant presence of informal transactions that should be regulated. The process of long-term rentals still results in frustration and lost time for both the owners and potential tenants. There has not yet been a significant technological disruption to conduct a new and better way to rent [51]. If we were to have a unified blockchain-based registry for real estate properties, the next logical step would be to facilitate the rental process by digitizing rent contracts through smart contracts. Blockchain and smart contracts might allow for automatic recurring payments, contract registration, and automatic payment of taxes, thus being an opportunity to promote the registration of rental agreements while reducing the black market [27]. It would bring several advantages to the landlord as the tenant's relevant background data, such as rental history, credit rating, and reviews, could be made visible and aid in the decision-making of signing a rental agreement.

Karamitsos et al. [36] conceptualized an Ethereum smart contract architecture for a real estate renting system. Only two entities were considered - contract owners (representing landlords) and users responsible for creating Ethereum wallets (representing tenants). The smart contracts serve as a way to establish a rental agreement between these two entities - the landlord initiates a contract by setting up the rental terms, and the contract is started as soon as a tenant signs it, meaning a rental agreement is formed. The tenant is then responsible for depositing funds in the smart contract through the payment function, which automatically sends the funds to the landlord. The contract also supports a termination function, in which the security deposit is returned to the tenants.

3.4 Investment Opportunities

The opportunities that Blockchain may bring to the real estate sector can even reach the investment world. Real Estate investment is simultaneously one of the safest and the most challenging forms of investing. This mostly has to do with the initial investment required to purchase property, and with the fact that real estate is notoriously difficult to liquidate, due to the challenges addressed before. More complications are added if we think about partitioning a property into smaller fractions. Just as companies issue shares, we can think of properties being divided into shares as well. Investors can buy and sell these ownership shares much like stocks in a company. An investment fund that follows this model is much more accessible due to its lower entry costs, but is currently inefficient due

to the slow liquidation.
A blockchain-based approach could potentially lift these challenges. Smart contract procedures could automate and simplify the exchange process, removing unnecessary steps and intermediaries. Through tokenization, these assets can be easily divided into smaller parts, allowing for fractional ownership and making investment more accessible.

Gupta et al. [29] described a workflow for tokenizing real estate within a public blockchain network, that allows for a more efficient and accessible way of real estate investment. This tokenization mechanism consists of a "legal trick" in which an intermediate business entity is used to indirectly tokenize a real estate asset. These business entities can be something like a Special Purpose Vehicle (SPV) or a Limited Liability Company (LLC) which are legal entities that have certain characteristics of a company, with their own assets, liabilities, and legal status. As such, they allow multiple investors to collectively acquire shares of it. By having a Special Purpose Vehicle own a single asset, the SPV can be sold itself, rather than attempting to split the asset between the various parties. The SPV would be tokenized and its shares would be distributed to the token holders. Buying a token would be equivalent to buying a share of the SPV that owns an asset, which in theory translates to acquiring a fraction of ownership over that asset [50]. This approach is currently one of the only reliable and legal ways to tokenize an asset and has been the means for most of the private companies to support ownership tokenization on their solutions, as we will discuss further ahead.

3.5 Related Hypothesis

As Saari et al. [53] verified in 2022 through an extensive literature survey, in the last few years the number of academic literature released about the potential uses of blockchain in the real estate industry has increased significantly. Still, very few of them present empirical evidence and practical examples of use cases.

Ali et al. [2] proposed a private permissioned blockchain smart contract architecture for land registration that would hypothetically replace Saudi Arabia's current infrastructure. The framework includes functionalities for facilitating the buying and selling process while providing a detailed transaction history and untampered information regarding a property. Being a private blockchain network, it would be managed by a central government authority that executes all types of contracts among the participants (buyers and land owners), with the participants responsible for submitting the required documents to complete the various operations. The network also includes other justice-related entities to help endorse the transactions, for instance by validating all the documents related to them. X.509 certificates and keys are generated for the various participants and the administrator of the network, and ideally a governmental entity would verify the keys and relate them to the identity of a citizen. The authors indicate that a proof of concept was implemented

using Hyperledger Fabric, however, the code was not made available.

Sen et al. [54] also proposed a smart contract-based permissioned blockchain framework in Hyperledger Fabric, for the Electronic Property Registration domain of e-Governance in India. The solution considered a set of peer organizations such as a Registration Authority (the one handling the Membership Service Provider (MSP)), a Land Authority and a Municipal Authority, and participants such as the seller, the buyer, mortgage agencies, and insurance agencies. The main objective was to ensure secure storage of the records by defining access rules for the various stakeholders. For this, the solution comprised the definition of three different channels with dedicated smart contracts - the first focused on property transactions, another for querying information and providing verification services to the involved agencies, and a final one exclusively dedicated to the private transactions between the three government organizations (the peers).

Alam et al. [1] proposed a three-phased blockchain-based strategy to modernize Bangladesh's inefficient and outdated land registry system. The current system relies heavily on physical documents managed by various authorities, leading to long and cumbersome processes susceptible to corruption and bribery. The authors recognize that one of the biggest challenges for the initial adoption of a blockchain system for managing title registry is to establish the necessary infrastructure. And so, for the first phase, a public blockchain network is proposed. The process would still be overseen by a government entity, however, regular people would be capable of being miners, thus alleviating the burden of rapidly constructing a comprehensive infrastructure. This first phase would only work as a timestamp logger of ongoing land handovers, ownership transfers, or financing events since it wouldn't be reasonable to think about moving all existing land records to the Blockchain system in an initial phase. The second phase introduces a small-scale hybrid blockchain network. Here, selected organizations equipped with high-capacity nodes would act as miners, and a private blockchain ledger would gradually incorporate the public blockchain data. This phase also incorporates multi-party PKI-based trusted communication for the involved parties and requires users to contact a Certificate Authority for public-private key pairs. The final phase is a full hybrid blockchain network, where even more public and private institutes with high computing capacities get involved in the mining process, and all documents are digitized and stored in a distributed file system like IPFS. Finally, a smart contract architecture illustrating the first phase is presented through an Unified Modeling Language (UML) Class Diagram, in which fractional ownership is supported, and includes functionalities for user creation, land registration, and assigning ownership.

3.6 Governmental Initiatives

Many governments have already shown interest in exploring what kind of opportunities blockchain technology could bring to real estate and land registration. We now present an overview of Proofs-of-Concept (PoCs) and Testbeds done worldwide in recent years.

3.6.1 Sweden

In the recent past, Sweden has tested applications for blockchain technology in their land registry system. The 2017 proof-of-concept refined and tested a blockchain-based solution for handling real estate transactions, to analyze the potential benefits of using smart contracts for land conveyance and mitigating existing problems mostly related to complexity, duration, duplication, and documents being physical, without the need for significant land agency disruption, complete IT infrastructure rebuilds, or full database redesign [5, 39]. The project was sponsored by a variety of entities such as ChromaWay (technology provider), Kairos Future (business), Telia (ID Provider), SBAB Bank, and Landshypotek (bank). The proposed solution consisted of a permissioned private blockchain network managed by either public or private entities, each with its conditions and authorizations. All the digital files relative to the ownership of land, mortgage deeds, and transaction processes are stored off-chain, while the blockchain network keeps an audit trail and handles the authenticity, the signatures, and the integrity of the whole process, by storing cryptographic hashes of the referred files. The solution took advantage of ChromaWay's Esplix technology - a system for exchanging signed messages - and their Postchain framework - an open-source relational blockchain solution - for the development of the consortium blockchain, as well as Telia's digital ID solution to handle user registration. Although the idea was for the Swedish Lantmäteriet (The Swedish Mapping, Cadastre, and Land Registration Authority) to store the blockchain with the proofs, the blockchain would also be stored and validated by other actors such as banks, real estate agents, buyers, and sellers. The results of the proof-of-concept could be seen in faster transactions, the development of a fully digital process, greater transparency for all parties, increased security, better redundancy of data, and the elimination of physical storage and other paper-based work such as traditional mail. The number of manual steps needed for a property transaction saw a dramatic decrease from 34 to just 13. Significant improvements were also pointed out in the process of mortgage deeds. From these takes, we can see that the project was a great success from the result point of view, however it ended up not being implemented. The reason for that lies in the fact that the Sweden law still does not allow for the use of electronic signatures for property transactions, and so further progress is expected once the digital signature restrictions are addressed [5].

Beyond the Swedish case, ChromaWay has also supported a few other initiatives in other legislations. Examples of those are the PoCs developed for Australia and Canada - both also hybrid solutions, but unlike the Swedish case, which focused on the complete

case of land conveyance, these projects are focused on more specific land administration processes [5]. The Australian case focused on the "Discharge of Mortgage Lien", which allegedly was an overly complicated process that included more steps than necessary, while the Canadian case focused on "Re-Assignment Reporting".

3.6.2 Estonia

Estonia has been proud to be a model to follow when it comes to promoting research in new technologies to reform traditional systems. It is a country where almost every bureaucratic task can be done online and when it comes to blockchain, titles itself as the #1 Country to use blockchain on a national level. In fact, Estonia had already started research on blockchain technology in 2008 – even before the release of the Bitcoin whitepaper, that first coined the term "blockchain" [20]. As of today, Estonia uses the KSI blockchain technology [21] to secure the integrity of government data and systems. That includes protection over services ranging from health records, law and court systems, banking, and, of course, land registry. While not the utopian scenario, it is stated that 'millions of lives and resources are saved as the potential manipulation of defense data or smart war machines is prevented using blockchain technology'.

3.6.3 Georgia

Disrupting the legacy systems becomes even more valuable in developing countries without a reliable track record of real estate ownership and land registry, where corruption might dominate and the integrity of official documents could be questionable. Projects of this kind may be the simplest and most efficient way to increase Gross Domestic Product (GDP) in the medium term [27].

Georgia's land registration and transaction process was infamous for being long and prone to corruption. Even so, its reputation has been changing drastically, as the government has been devoted to modernizing its systems. In 2016, it was announced that a collaboration between the government of Georgia and BitFury - an American provider of Bitcoin blockchain infrastructure - had taken place, aiming to design and pilot a private permissioned blockchain operated by the National Agency of Public Registry (NAPR) and anchored to the Bitcoin Blockchain through a distributed digital time-stamping service [55]. The pilot was a success and so, later in 2017, the system was expanded, allowing blockchain technology to facilitate the purchase and sale of land, reducing the transaction costs to about 0.1% of the property value [56, 58]. The project has been completed and made available to the public, making the Republic of Georgia the first government to use blockchain technology to store and validate land registry information. Following the constant updates and renovations, Georgia ranked fifth in the world for ease of registering a property, according to The World Bank Doing Business Report, in 2019 [67].

3.6.4 Honduras

Honduras is another case of a developing country trying to improve its land registration system through blockchain technology. This one, however, ended up not succeeding. The land registration system in Honduras faces significant challenges and inefficiencies. Title fraud and corruption are usual, ownership information is not properly recorded and is easily tampered with. According to USAID in a 2018 estimate, only 14% of Hondurans legally occupy properties [65]. In 2015, the government of Honduras reached out to Factom - an American blockchain technology company - to discuss the possibility of developing a new system for land registry. Factom proposed a blockchain-based solution layer to maintain a permanent, timestamped record of a land transfer on top of Bitcoin's blockchain [10]. By this time, numerous press reports could be found about this initiative, however, the Honduran Government never made any public comments about it, and no further developments were ever known. In 2018, Castellanos et al. [10] conducted an interview with the liaison between Factom and the Honduran government, to further investigate the status of the project. From the interviews, it was learned that there were several difficulties imposed by the government, some of which were related to the project happening near an election cycle, leaving government officials reluctant to introduce any major changes that could be used as a threat to the sitting government. The project was halted in mid-2017 due to the impending presidential elections and has not been reactivated since then [10].

3.7 Commercial Solutions

3.7.1 Propy

In recent years, Propy has caused an uproar, announcing to conduct "the first real estate transaction ever sold and transferred on the blockchain" [47]. This accomplishment is certainly impressive, however, there's a slight catch. Propy's mission is in fact for government bodies to recognize its registry as the official ledger of record such that property transactions made on the Propy platform are acknowledged as legal transfer of the property ownership. But as of right now, it mostly functions as a platform that complements and integrates with existing land registry systems. Propy's platform allows for the digital upload and signing of traditional documents such as real estate ownership agreements and mortgage deeds [48]. This simplifies and eases the process of a real estate transaction since the process can be done mainly digitally, saving hours for everyone involved in the transaction. All the traditional paperwork is still on board, so this platform does not replace lawyers or intermediaries but instead provides them with tools to facilitate document sharing and signing, notifying all parties involved. Additionally, Propy provides multiple ways to carry out the transaction itself, whether from a traditional deal, from using cryptocurrencies, or by tokenizing a real estate asset and auctioning it. To make this possible, Propy mirrors government records in its blockchain network built on top of

Ethereum and uses IPFS to store all the required files and documents. PRO tokens act as the main currency for crypto transactions and are built on top of the ERC-20 standard for Ethereum fungible tokens. Tokenized Real Estate properties are represented by a Propy-developed NFT which is allegedly capable of tokenizing any property in the United States. Transferring the possession of this NFT acts as a proxy for the purchase of the real estate since Propy has developed proprietary legal paperwork to encapsulate property rights into a US-based legislation entity [48]. Finally, Propy's smart contracts act as payment escrows in the transaction. So going back to the first statement, it wouldn't be fair to say that the real estate property was entirely sold via blockchain, but instead that a blockchain platform was used to facilitate the traditional transaction process.

3.7.2 Unlockit

Unlockit is a Portuguese startup aiming to revolutionize the Portuguese real estate market, which is known for being slow, outdated, and lacking transparency. To address these challenges, Unlockit plans to use blockchain technology to unify property-related data, which is currently fragmented across multiple domains owned by various stakeholders. In practice, Unlockit intends to leverage Digital Asset's Daml Canton interoperability protocol to create a privacy-enabled network of networks. In this system, Unlockit will establish a permissioned network where participation is granted to those with a participant node, which will be distributed and facilitated by Unlockit. The first step is to onboard private institutions (mostly real estate agencies) onto this network, but the ultimate goal is to also bring all public institutions together, creating a cohesive ecosystem. Within this ecosystem, Unlockit aims to digitize all information related to real estate assets, so that eventually processes such as ownership transactions, issuing and signing documents, and even making payments can be made fully on-chain. Intuitively, the information within this ecosystem will encompass both private and public data. And this is where the privacy control features of Daml shine. As discussed in subsection 2.9.6, Daml will enable the configuration of which participating organizations can access specific information within this universal system and what information can be publicly accessed by end users, such as citizens and clients. This is, however, their vision for the future. Unlockit's current commercial product is a digital governance platform that facilitates agile digital signatures and the digital transfer of documents. The blockchain component of the platform is still under development. Unlockit is currently developing smart contracts to facilitate pre-sale agreements, and thus it is working on digitizing documentation to represent it on its blockchain, ensuring it can be verified during the buying and selling process. The next step is integrating the smart contracts with banking systems and fund transfer mechanisms. But despite these advancements, what they currently have is still far from the final product. Unlockit is in an exploration phase, studying how to enter the market effectively and how to incentivize governmental entities and involved stakeholders to adopt such a disruptive system.

3.7.3 RealT and Binaryx

RealT and Binaryx are both companies that take advantage of the previously mentioned approach described by Gupta et al. [29] - the use of intermediate legal entities in order to indirectly tokenize a real estate asset. This "legal trick" comes down to creating a business entity such as an SPV or an LLC for each real estate property, and legally assigning it as the property's owner. Then, fungible tokens are created each representing a share of that business entity ("RealTokens" for RealT [50] and "bAsset tokens" for Binaryx [7]), which can be bought and sold through their marketplace platform. Buying a token of a real estate property is equivalent to buying a share of the business entity that owns that property, which in theory translates to acquiring a fraction of ownership over the asset [50]. But in practice, it is the business entity that is being tokenized, and not the actual underlying asset [29]. This means that RealT and Binaryx essentially provide an investment platform for real estate, where investors can purchase fractional shares for as little as US$50. This approach effectively removes the primary obstacle to entering the real estate investment market, which is the high cost of entry. As to property management, since a property listed on these platforms can grow to have thousands of owners, it is unrealistic to expect all of them to coordinate proper management decisions. Instead, RealT and Binaryx offer services to upkeep the property and manage all landlord responsibilities, while the token holders only have to worry about voting for basic operations like raising or lowering the rent cost [7, 50]. This is usually done through a Decentralized Autonomous Organization (DAO) implemented with smart contracts. RealT's system architecture is built on top of Ethereum, and the RealTokens are a branch of the ERC-20 standard for digital tokens. Dai stablecoin is used to distribute rental incomes to the RealToken holders, ensuring that these are not subject to the volatility often associated with other cryptocurrencies. IPFS is also used to provide all RealToken owners with access to their relevant documents [50]. The implementation details of Binaryx are still not publicly available since the company is still at an early stage.

There are other companies that proposed similar approaches to RealT and Binaryx. Laprop [40] is a recent one, however, there is still not much to be found about it other than the first whitepaper released in 2022. REX Protocol [52] is another initiative that appeared in the same time period, and similar to Laprop, no recent developments were found. Atlant was one of the most promising initiatives in 2017 [4], but information regarding the project seems to have stopped, so it is likely that it failed.

3.7.4 Rentberry

The rental market is also already being explored by companies like Rentberry, which describes itself as a "decentralized long-term rental ecosystem that uses blockchain technology to make the rental process less costly and more convenient and secure" [51].

Rentberry's platform can be used to complete most rental tasks like searching for properties, making offers/bids, negotiating terms, e-signing contracts, and paying rent. The platform runs on top of Ethereum, and its transactions are made using BERRY tokens (a variant of ERC-20) as the currency, which can be used for a variety of other services like property promotion and marketing or hiring third-party service providers, such as house cleaners, handymen, plumbers, electricians, etc. The platform is already fully operational and accessible to the public via their website. Rentberry serves as a good example of how there is room for improvement in the current rental process, and how blockchain technology could help overcome many of its current challenges.

4

System Model and Architecture

Many of the solutions presented in the previous chapter are hybrid approaches in the sense that they propose to keep the legacy land registration infrastructure as is and aim to complement it in some way by using blockchain technology. They deliver the benefits of smart contracts whilst minimizing risk and resistance by ensuring minimal disruption. These approaches are very valuable and are the most likely to be implemented in the near future, as complete tokenization of property within any jurisdiction is probably still several years from becoming a reality [6].

The goal of this work is to imagine and conceptualize the architecture of a system that would rely on blockchain as the foundational infrastructure to store any land related data, by having a blockchain network replace the central databases and store the actual records. The idea is that for each existing contract and documentation, there will be a smart contract to replace it. This smart contract will contain verifiable and codifiable logic related to the contract, allowing for the automation of several processes. However, a premise we considered when designing the model was 'what is the reality today'. We do not want to alienate the current system. It is true that the introduction of smart contracts has great potential for expanding functionality and making tweaks to legacy processes, but proposing such extensive changes in the context of a thesis would be impractical. Reformulating and introducing a new model is extremely complicated due to the various attack vectors that exist. It is inconceivable without thorough consideration and testing. Therefore, the objective is to reflect what currently exists while incrementally improving and modernizing the system through the use of blockchain technology.

4.1 Project Scope

4.1.1 Desired Characteristics

Before jumping into the details of the proposed solution, it is important to explain why a permissioned consortium blockchain network is preferable for the context of the project, compared to public ones. The first reason is that land registry is by nature a sector that must be regulated by a country's government. Additionally, it involves many entities that play indispensable roles in the process, each controlling a significant part. This complexity and the need for regulation make a consortium blockchain particularly suitable, as it allows for shared control and governance by multiple organizations. Furthermore, many of the characteristics intrinsic to public blockchains are not only unnecessary but also undesirable in this context.

1. **Non-Anonymity** - Public blockchains grant anonymity for all network participants. However, in the transaction process of a property, both parties must be unambiguously defined and identified;

2. **Need for Privacy** - Certain data and documents regarding property information or even transactions shall be kept private from certain stakeholders or the overall public;

3. **Need for Network Administrators** - There may be needed mechanisms for error correction and ownership recovery in case incorrect or fraudulent data is discovered, possibly in the form of multi-signature protocols [28]. Some permissioned environments also allow members to come to an agreement and alter previous blocks [18];

4. **Infrastructure Management** - The management of the infrastructure for a system of this kind is a critical decision. Controlling the infrastructure means controlling the real estate assets of a nation. Utilizing public infrastructure for this purpose poses significant risks and is generally not advisable;

5. **Security Concerns** - Although highly unlikely, public networks are susceptible to 51% attacks and future threats from quantum computing;

6. **Network Traffic and Congestion** - Transactions in public blockchains are often affected by network congestion, which not only increases the time it takes to confirm them but also leads to higher transaction fees;

7. **Currency** - Public blockchains rely on their own native cryptocurrency for conducting transactions. This dependency would introduce volatility and transaction fees, thus making a part of the transaction value leave the country. It is preferable to use the country's currency for transactions, given the sector's direct linkage to the national economy;

8. **Storage Considerations** - Public blockchains are particularly unsuitable for environments that require storing large amounts of data, as their vast node number comes with bandwidth limitations and high transaction costs. In permissioned environments, this challenge is mitigated.

Finally, permissioned blockchains would also provide flexibility in making transactions comply with country-specific legal and tax requirements [27] while providing faster and more efficient transactions with reduced latency.

4.1.2 Roles and Stakeholders

It is vital to define which stakeholders will be considered in the proposed solution, both those responsible for managing and maintaining the system (peer organizations) and those that will use the system as end users. Figure 4.1 presents a diagram illustrating these entities, and how they engage with the system in terms of what operations they perform within the blockchain network:

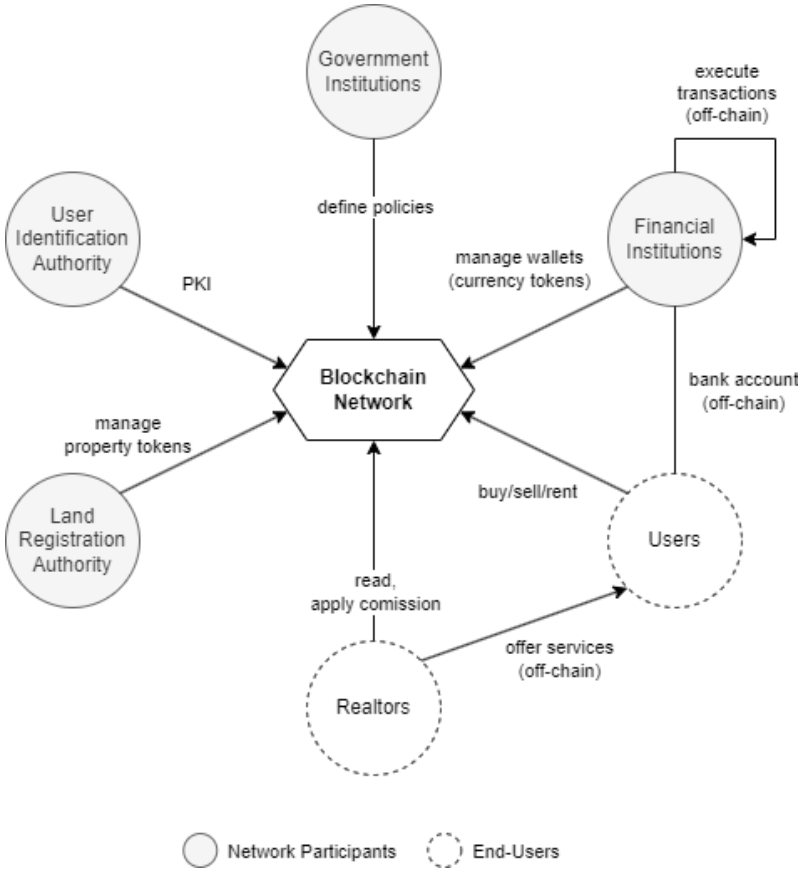


Figure 4.1: Domain Interactions Diagram

Users - The typical end users of the system, representing the average citizens. They might own properties and wish to sell or list them for rent, or they may be seeking to buy or rent properties.

Realtors - Institutes or individuals that aid the users in the process of buying or selling a property, with the goal of facilitating the transaction process. They provide expert guidance on market trends, property valuation, and legal requirements.

Land Registration Authority - The entity responsible for validating the property records, guaranteeing that they comply with the actual physical properties and location of the land. Within the blockchain network, it is responsible for minting and managing the property tokens.

Government Institution - The government body that enforces the country's policies and regulations, ensuring that the system adheres to legal standards.

Financial Institutions - Financial institutions such as banks hold the users' actual funds and are responsible for relating these funds to the users' accounts on the blockchain network, guaranteeing that the transactions made on-chain are mirrored in the real world.

User Identification Authority - The entity responsible for providing and managing the PKI, by issuing the certificates and linking them to the users' real-world citizen identities.

4.1.3 Core Functionalities

4.1.3.1 Fractional Ownership

We must assume that a property may belong to multiple entities at the same time, being divided into either dependent or autonomous fractions. Dependent fractions are percentual, meaning that a single property like a household could, for instance, be owned 30% by an entity and 70% by another. Autonomous fractions refer to independent properties that compose a property cluster - an apartment is an autonomous fraction of a building. An autonomous fraction can be further divided into dependent fractions.

4.1.3.2 Document Storage and Emission

We intend to explore how the blockchain system could be used to store any type of data related to a property. This not only includes property metadata, such as address, geographic location, size measurements, maps and blueprints, but also any kind of document that may be issued for it. Depending on legal requirements, various documents might be necessary to advance the property sale process, ensuring that the property complies with specific regulations and is properly legalized.

4.1.3.3 Land Conveyance

The main focus of the solution is to support land conveyance. This refers to the process of transferring ownership of a property from one user to another. Due to fractional ownership,

we must also consider the possibility of buying and selling individual fractions, instead of the whole property. Associated with the sale of a property, there may be a commission taxed by a real estate agent, if one was hired to facilitate the deal process.

4.1.3.4 Leasing

We also plan to demonstrate how smart contracts can be used in leasing, since they can be used to enforce verifiable clauses commonly found in rental agreements, and automate the payment schedules. These smart contracts could handle conditions such as rental due dates, security deposits, and penalties for late payments.

4.1.3.5 Mortgage Loans

Mortgage loans, or simply mortgages, are among the most common procedures when purchasing a home. This is because it is rare for a buyer to have the capacity to purchase a property outright. For this reason, it would be interesting to demonstrate how loans could be integrated into the system and how the amortization logic could be recreated through smart contracts.

4.2 Development Framework

Following the considerations discussed in section 2.9, we decided to go with Hyperledger Besu as the blockchain development framework for this work. We picked it over other contenders mainly because of its versatility and general-purpose nature. Being an Ethereum client, Hyperledger Besu benefits from Ethereum's vast ecosystem of development frameworks and comprehensive documentation.
Realistically speaking, in a production environment, it would be preferable to pick a more specialized framework, especially in a domain like real estate and land registry. Daml Canton is an example of a tool specifically designed for drafting complex multi-party agreements and operating over digital assets. However, a solution using Daml Canton would focus on interoperability and privacy of information, organizing data so that it can integrate various global processes while keeping certain information private from some participating entities. This aspect of privacy and interoperability is critical in production settings. But for the demonstrative purposes of this thesis, we do not intend to focus on these aspects as heavily.
Thus, the adaptability and applicability of Hyperledger Besu make it a compelling choice. We aim to provide an architecture that may be used and followed even when using other technologies. The other contenders, like Fabric and Corda, were also viable alternatives. However, they are somewhat unconventional in sense that chaincode (Fabric) and CorDapps (Corda) do not adhere to traditional generic smart contract architecture. For instance, in the case of R3 Corda, the developers themselves have noted that "Corda is both a blockchain and not a blockchain" [49]. Choosing Corda would imply building a

Corda-centric solution and architecture. Therefore, given the scope and aims of this thesis, Hyperledger Besu's compatibility with traditional smart contract architecture and its long-term community support within the Ethereum ecosystem make it the most suitable choice for this work.

4.2.1 Gas Considerations

Being an enterprise blockchain framework, it is possible in Hyperledger Besu to configure a network without the use of gas. Still, it provides the option to retain ETH and apply gas to transactions even in a permissioned setting. On one hand, it might seem logical that gas would not apply in a private environment where access is naturally restricted and processing transactions is not as resource-intensive, as discussed in subsection 2.5.1. Removing gas, however, would imply discarding the anti-spam measures that gas typically provides [23]. The decision to maintain or remove gas from a private network is not as straightforward.
In some cases, using gas could actually come in handy. One example is to use it for taxation purposes, representing a cost associated with using the network infrastructure, where the collected fees could be used to fund network maintenance or development. In the context of this work, we can imagine that in a production environment it would be desirable to have various types of fees associated with real estate transactions or the issuance of documents, as occurs in real life. However, the value of these fees would naturally vary depending on the type of service, and not on network congestion. Thus, it would be better to calculate these transaction taxes within the smart contract logic, instead of using gas for this purpose.
The network configurations for the proposed solution deploy a permissioned network without gas. But simply removing gas from the network would pose significant security risks. It is mandatory to make security considerations mainly regarding flooding and infinite smart contract loops.

4.2.1.1 Smart Contract Infinite Loops

An infinite loop occurs when a smart contract is written or interacts in a way that causes it to execute its instructions repeatedly without a termination condition. This can happen either accidentally, due to a programming error, or maliciously, when someone intentionally creates a contract that will continue to run to disrupt the network. In the Ethereum Mainnet, once the gas allocated for a transaction runs out, the contract execution stops. On a permisisoned environment, an alternative to gas is to leverage **permissioning** mechanisms that allow us to restrict which entities can deploy smart contracts. This approach may or may not rely on some trust assumptions regarding the entities allowed to deploy contracts, depending on the implementation. This was the approach we adopted in this work, and will be elaborated further in the next section.

4.2.1.2 Flooding

Flooding refers to overwhelming the network with a high volume of transactions, which can lead to congestion and degrade the network's performance. In Ethereum Mainnet, the gas price increases during high demand, discouraging malicious submissions. To address the risk of flooding in a network without gas, we assume for simplicity that client behavior is controlled using off-chain tools such as firewalls and access control mechanisms. Alternatively, it could be feasible to limit the number of transactions a single account can submit per unit of time. However, setting such limits would require careful consideration to avoid accidentally restricting legitimate activity. Mechanisms such as the bundling of operations could be implemented to prevent this but would introduce complexity. Implementing techniques such as exponential back-off could help managing transaction submission rates, contributing to network stability.

4.3 Permissioning

Permissioning is the mechanism through which it is possible to define which accounts and nodes can do what within the system. It let's us define administrators, and restrict actions accounts can take. It is through permission logic that we can determine which entities can instantiate smart contracts, mint tokens, invoke functions, an so on. Defining who can instantiate smart contracts is particularly relevant given the context of this work, since defining which smart contracts will be used is essentially defining the rules within the system to which all users are subject. It translates to defining the law in the real world.

4.3.1 Permissioning Types

Each permissioned blockchain framework tends to have its own unique method of ensuring data privacy among subsets of the network participants. Hyperledger Fabric, for instance, has integrated permission mechanisms within its own PKI and has a native concept of peer organizations. On the other hand, Enterprise Ethereum Clients that comply with the EEA, such as Hyperledger Besu, need to ensure ways to control which nodes may join the enterprise blockchain network and to restrict the ability of individual accounts or nodes to perform specific functions. For instance, an enterprise Ethereum blockchain might allow only certain accounts to instantiate smart contracts [44].
Besu provides two main types of permissioning: Local permissioning and Onchain permissioning.

Local Permissioning - Local permissioning works at the node level. Each node in the network can be configured with a file that specifies the whitelist of accounts and other nodes that node may connect to.

Onchain Permissioning - Onchain permissioning works through at the smart contract level. Permission smart contracts shall be deployed on the network and must store and administer the allowlists for nodes, accounts, and admins. This method allows for more complex and secure permission mechanisms. It also enables all nodes to read and update permission configurations from one single source.

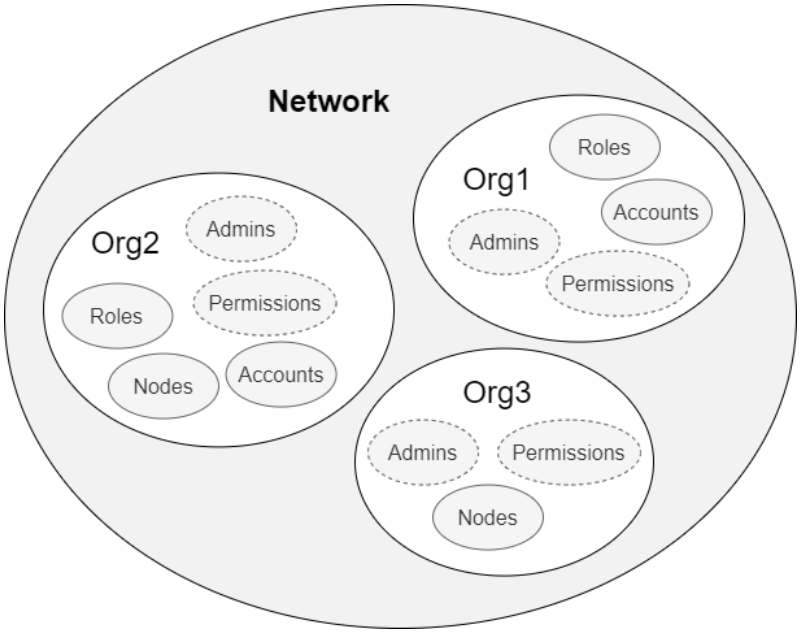


Figure 4.2: Permissioning Model Diagram

4.3.2 Permissioning Model

The developed solution uses onchain permissioning and integrates a generic permissioning model that is primarily focused on organizations and roles. Fundamentally, the system consists of participating organizations that represent the various stakeholders contributing to the system for a specific purpose. As described in subsection 4.1.2, examples of such organizations include government institutions, financial institutions, the land registration authority, the user identification authority, and others that might become relevant in the future, such as the courts or evaluators.

Each organization must be restricted on the type of actions its members have permission to perform within the system. These permissions include deploying contracts, minting currency tokens, registering new real estate properties, or offering loan services, among others.

Within each organization, user accounts are registered, and each must be associated with a specific role. Each role is linked to only one organization. The need for different roles within an organization is justified by the necessity of having accounts with completely different permissions. Moreover, organizations can designate which users are admins, indicating who represents the organization and can vote on its behalf in decisions requiring

the consensus of multiple system organizations.
Additionally, nodes may also be registered within an organization. However, there may be a need to allow organizations to participate in the system without having control over their infrastructure, thus making this another configurable permission.

Questions may arise concerning how end users, citizens without regulatory, executive or authorized intermediary roles, fit into this scheme of organizations. The existence of the previously mentioned User Identification Authority is specifically for this purpose. End users, which are independent of other stakeholders, shall be registered under a governmental body responsible exclusively for creating and managing user accounts and associating them with the individuals' civil identities. This organization should not have additional network permissions beyond those necessary for registering and managing accounts.

4.4 Privacy

Another crucial feature of permissioned blockchains is the ability to keep data private from some participating entities. Sometimes, we might want to have processes exclusive to a subset of participants that others cannot see or simply ensure some data confidentiality. In the context of land registry, it is reasonable to assume that we may want to keep some property-related data private from some participating organizations that might use them for undesirable purposes.
In Ethereum development, confusion might arise regarding Solidity's contract-level private visibility specifier. The private visibility specifier means that a variable is only visible within the contract where it is defined and not in derived contracts. However, that doesn't mean the actual data is private. On Ethereum, all data, regardless of its visibility specifier in code, is transparent. If someone knows how to navigate Ethereum storage, they can access this data. Enterprise permissioned Ethereum clients provide alternatives for this. EEA-compliant Ethereum clients such as Hyperledger Besu typically offer privacy through **private transactions**. Private transactions refer to the ability to keep transactions private between a subset of participants. They are fundamentally similar to public Ethereum transactions but instead of being broadcast to the entire network, they are transmitted through direct point-to-point distribution to only the intended recipients. To implement this kind of privacy at transaction level, Besu shall be deployed along with a private transaction manager like **Tessera**.

ConsenSys Tessera [13] is an open-source privacy transaction manager that communicates with privacy-enabled Ethereum clients through an API that uses two-way SSL and TLS certificates. Each Besu or Quorum node that sends or receives private transactions must be associated with a Tessera node [32].

When a private transaction is initiated, it first passes from the Besu node to the

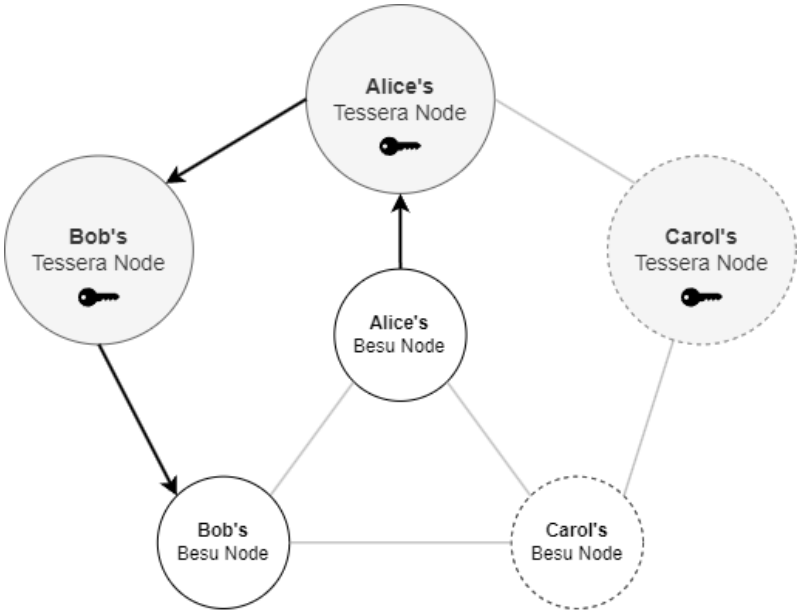


Figure 4.3: Tessera-Besu Interaction Model. Source: Adapted from []

associated Tessera node. The Tessera node then encrypts the transaction data and conducts direct point-to-point distribution to transmit the transaction to the intended **privacy group**, ensuring only those can decrypt and access its details. Privacy groups are clusters of nodes that are identified within Tessera by a unique group ID. In the context of a Besu network, these groups maintain a private state that is inaccessible to nodes not participating in it. The private state is kept separately from the public world state, and thus communication between these states is very restricted in order to guarantee data consistency to every network node. A contract deployed within a privacy group:

- Can read and write to contracts within the same privacy group;
- Can read from (but not write to) public contracts;
- Can neither read nor write to contracts deployed in a different privacy group.

The proposed solution supports private transactions with Tessera, and thus it is possible to make private transactions. However, we should note that the rest of the implementation doesn't make use of them, since in practice there's not much to demonstrate. Still, it is an important feature that would have great value in a production setting.

4.5 Application Stack

We now present the complete application stack that was utilized throughout this project. In addition to the technologies previously discussed, a web client application was developed to facilitate the demonstration of the proposed solution. It was developed in *React* with *TypeScript*, and uses the *ethers.js* library to interact with the network. In addition, *Metamask* wallet service was used to facilitate authentication and account management.

The development, compilation and deployment of smart contracts was facilitated by the use of Hardhat, a development environment that offers an extensive suite of tools for building Ethereum applications, and also includes smart contract testing capabilities. Finally, Hyperledger Caliper was chosen as a benchmarking tool to evaluate the performance and scalability of the smart contract system.

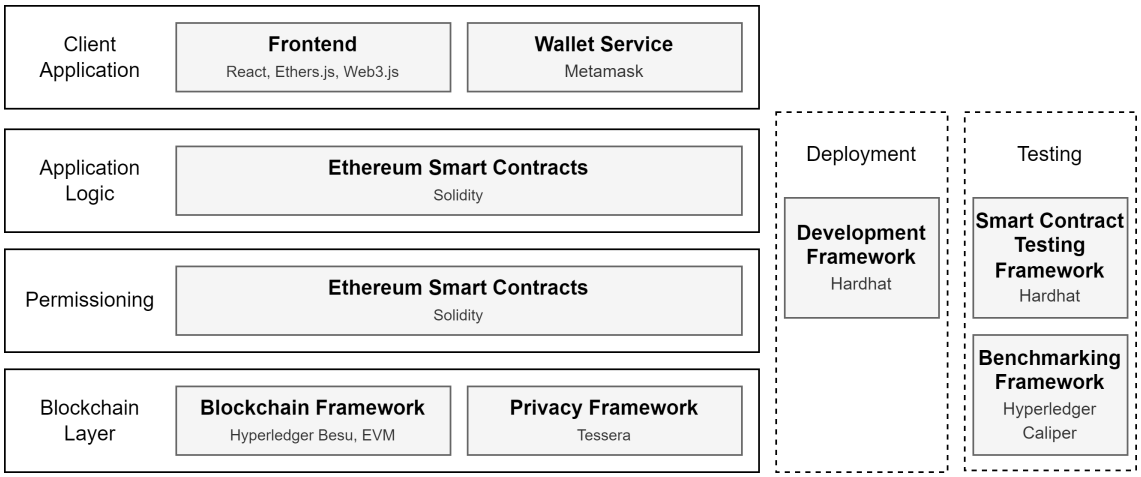


Figure 4.4: Network Topology Stack

4.6 System Metamodel

Appendix A includes a UML Class Diagram that serves as a metamodel for the proposed solution's architecture. This diagram provides a high-level overview of the system's architecture, primarily focusing on the data that the system will store, how this data should be organized into different entities, and the functions associated with these entities. The relations between these entities illustrate how they are interconnected and associated, which is essential for understanding the overall structure and functionality of the system. Additionally, Object Constraint Language (OCL) [45] restrictions were defined to specify the system's behavior for each of the stated functions. OCL is a logic language commonly used to write detailed specifications and constraints for UML models. It allows for the precise definition of rules that the system must adhere to, ensuring that the system's operations are consistent with the intended design and requirements.

It is important to emphasize that this metamodel does not directly reflect the smart contract architecture of the final solution. Rather, it should be viewed as a guideline that provides information on how the system should be structured and operate. The actual implementation, especially in the context of smart contracts, might have specific nuances and complexities that are adapted to the practicalities of the technology. However, it would be fair to say that, independently from the implementation details, the final solution should comply with the metamodel.

5

System Implementation

In this chapter, we delve into the implementation details of our system and explain the rationale behind each design decision. We also discuss alternative ideas that were considered, comparing the advantages and disadvantages of each. It should be noted that, to facilitate understanding and for the benefit of the reader (since code can be quite challenging to interpret for those not involved in the project), we do not display the actual code in most instances. Instead, we present interfaces for each of the proposed smart contracts, explaining the logic provided by each operation. For those eager to learn about the more specific details of the implementation, the code is available for consultation and can be found in the official GitHub repository of the *Blockopoly* dissertation, available at *rmarede/Blockopoly*.

In general, the presented smart contracts were designed with a strong commitment to adhering to established best practices and design patterns found in Ethereum development. Our primary focus was creating a modular solution supporting continuous improvement and integration of new features. However, due to the project's demonstrative nature, in which the main goal is to explore the capabilities and potential of smart contracts in the context of real estate transactions, and because the proposed solution is intended for a private and dedicate environment, gas efficiency was not taken as a primary concern.

5.1 Network Configuration

The GitHub repository contains all the necessary configurations for deploying a privacy enabled Hyperledger Besu network, that complies with the considerations discussed in the previous chapter. These configurations include Docker container deployment files, custom scripts for generating cryptographic material for the nodes, network configuration files related to protocol versions, consensus algorithms, and finally scripts for smart contract deployment and testing.
To simplify the deployment phase for testing and demonstration, several auxiliary scripts have been developed that automate this complex and time-consuming process, reducing

it to just a few steps. Additionally, scripts are also provided for the automatic deployment of all the smart contracts that will be discussed in the following sections, along with their initialization.
The step-by-step tutorial on how to use those scripts, as well as an in-depth analysis of the project structure are both additional artifacts of this work. These are available for consultation through the *README.md* files across the various directories of the repository. The network utilizes the Istanbul Byzantine Fault Tolerance (IBFT) consensus protocol with no gas fees or limits.

5.2 Contract Name Service

The Contract Name Service (CNS) is an auxiliary smart contract which was designed to facilitate communication among the various developed smart contracts. It is an example of the Contract Registry Pattern [42], which consists of having a registry contract storing the addresses of multiple versions of one or more smart contracts. In the context of this project, the CNS primarily functions as a service name resolver. This means that when a smart contract needs to invoke a function from another smart contract, instead of having the address of that smart contract predefined or hard-coded, it first consults the Contract Name Service to obtain the latest address for the desired service. This setup offers several benefits to the project. Firstly, it simplifies the initial network setup phase, as it eliminates the need to configure each contract with the addresses of the various contracts it depends on. By registering the address of each contract just once in the CNS, and then simply passing the CNS address to each contract, configuration becomes streamlined and more manageable. Moreover, the Registry Pattern can also see usage in smart contract upgrading, particularly in cases where the contracts to be upgraded do not contain internal state. This is because a new contract will not retain the internal state of the previous version unless a specific migration mechanism is implemented.

```
1 interface ContractNameService {
2     event ContractRegistration(string indexed name, address addr, uint version);
3     function getContractAddress(string _name) external view returns (address);
4     function getContractVersion(string _name) external view returns (uint);
5     function setContractAddress(string _name, address _address) external;
6     function isRegistered(address _address) external view returns (bool);
7     function isRegistered(string _service) external view returns (bool);
8     function getContractHistory() external view returns (ContractInstance[]);
9 }
```

Figure 5.1: Contract Name Service Contract Interface

It should be noted that using this approach adds an additional step in smart contract communication, which naturally makes the transaction execution a bit slower. In public

networks, it is often seen as a serious drawback, as public solutions prioritize gas optimization, and upgrading smart contracts is in most cases an undesirable feature. In private solutions, the additional overhead may not be significant enough to justify avoiding this approach, considering the benefits explained before. Resource savings are less substantial since the number of nodes is much smaller. Still, it is one feature whose worthiness should be deeply considered in a production environment.

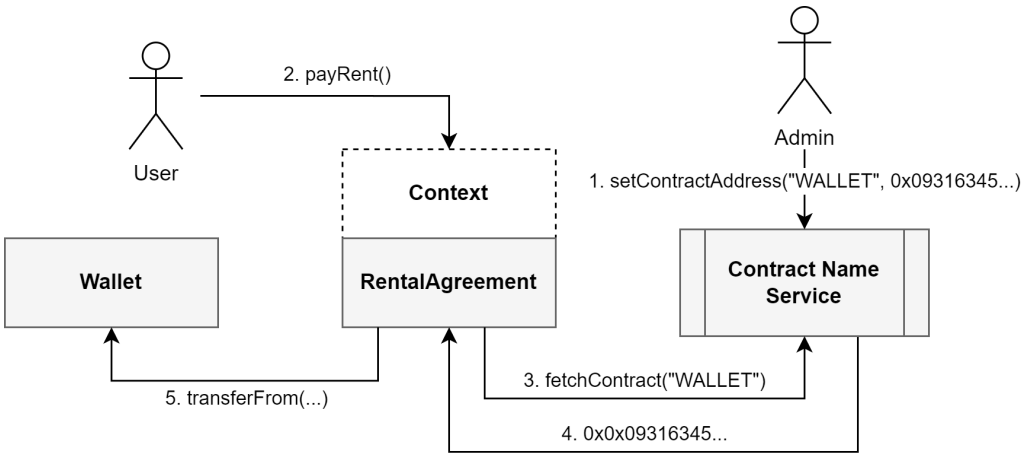


Figure 5.2: Contract Name Resolution Flow

5.3 Multi-signature Accounts

Multi-signature (or just multisig) wallets or accounts are smart contracts that aim to increase security by requiring multiple parties to confirm transactions before their execution. They are appropriate for scenarios in which sensitive actions can only be executed if a set or a subset of predefined entities agree to do so. These transactions can be anything within the blockchain ecosystem - transferring some currency to another account or contract, calling other contracts' functions, and even deploying contracts.
Multisig implementations typically follow an architecture with the following operations:

SubmitTransaction - Any participants defined within the contract may submit a transaction proposal. The transaction metadata will be saved within the contract and made public to the other participants, but the transaction will not be executed until enough confirmations are reached.

ConfirmTransaction - Other participants may look for submitted transactions and confirm them if they agree with their execution.

ExecuteTransaction - Once enough participants confirm a submitted transaction according to the confirmation policy defined within the contract, it shall be executed. This confirmation policy could require the confirmation of every defined participant or only a subset, such as most.

It is also important to note that transactions executed by the multisig account are executed by the contract itself, meaning the address sending the transaction is the multisig contract's address. This means that if the transaction submitted involves transferring some currency or some asset, the multisig contract needs to have enough balance or allowance for the asset being transferred.

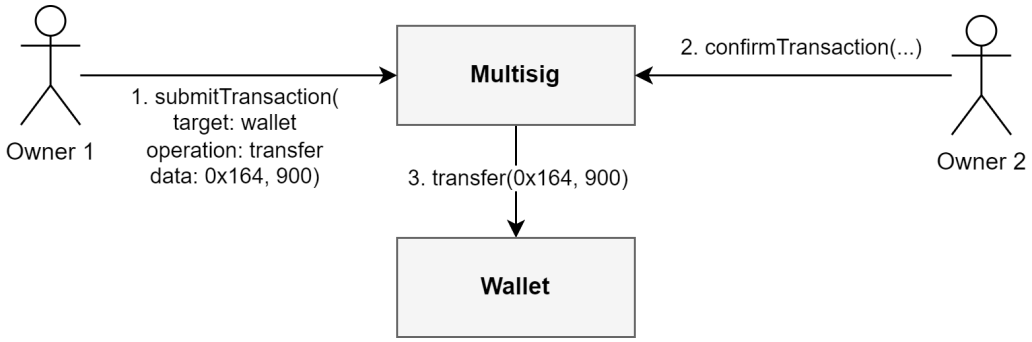


Figure 5.3: Multisignature Account Flow Diagram

Various implementations of multisig wallets can be found online, such as the well-known Gnosis Multisig by OpenZeppelin [46]. That said, for our specific requirements, we found the necessity to build different and derived versions of them. Multisig accounts were used in several instances along the project, in some cases in ways that are not typically seen. The different versions implemented are enumerated in the next sections, and more details on how they were using for the solution in particular are elaborated ahead.

The multi-signature pattern is at the **very basis of the proposed architecture**, and hence we strongly emphasize the importance to properly understand how it works and what are its advantages.

5.3.1 WeightedMultisig

The *WeightedMultisig* is a multi-signature smart contract in which it is possible to define how much the vote of each different participant weighs to the confirmation of the transaction. This means that a participant with a voting weight of 2 will have double the voting power of participants with a weight of 1. The contract also supports functions to add, remove, and transfer voting weights between the participants. The goal for this contract was to make its logic simple but highly versatile since it will be applied to different parts of the project with different requirements and characteristics.

```
1  interface IWeightedMultiSig {
2      function submitTransaction(address _destination, uint _value, bytes _data)
           external returns (uint);
3      function confirmTransaction(uint _transactionId) external;
4      function executeTransaction(uint _transactionId) external;
5      function getTransaction(uint _transactionId) external view returns (address,
           bytes, bool);
6      function isConfirmed(uint _transactionId) external view returns (bool);
7      function hasConfirmed(uint _transactionId, address _participant) external
           view returns (bool);
8      function getConfirmationCount(uint _transactionId) external view returns
           (uint count)
9      function getParticipants() external view returns (address[]);
10     function transferShares(address _from, address _to, uint _amount) external;
11     function addShares(address _to, uint _amount) external;
12     function removeShares(address _from, uint _amount) external;
13 }
```

Figure 5.4: WeightedMultisig Contract Interface

5.3.2 SelfMultisig

The *SelfMultisig* is a relatively generic multi-signature smart contract which is configured to only be able to call functions within the smart contract itself. Alone, the *SelfMultisig* smart contract has no use, since it got no other functions besides the basic set for the multisig pattern. However, its primary purpose is to be inherited and expanded upon, adapting it to specific circumstances. Multi-party agreements, such as sale and rental agreements, often require actions to be executed only when all involved parties agree. In the context of smart contracts, this requirement translates well into the multi-signature philosophy. However, making them traditional multisig wallets would let the parties use them for numerous other activities that may not be related to the logic of that specific contract. The *SelfMultisig* smart contract addresses this issue by allowing parties to agree on and execute only the functions defined within the inherited contract. By extending the *SelfMultisig*, developers can define the necessary functions, thereby limiting multisig operations to those explicitly designed for the contract's intended purpose. This approach ensures that the multi-signature agreement remains focused and relevant to the specific context it was created for.

5.3.3 OrganizationVoter

The *OrganizationVoter* is another instance of the multisig pattern which is not so generic as it is designed to fit the organizational context of the project. As we will elaborate further in the next section, there are many privileged operations within the system that should only be invoked through a voting mechanism between the various organizations (for instance, smart contract deployment). We opted to do that with multi-signature accounts. This contract changes the capabilities of standard multi-signature contracts to

require a consensus among participating organizations, instead of the typical predefined account addresses. Its interface offers operations similar to standard multisig pattern implementations.

5.3.4 Multisig Design Considerations

The main drawback of using the presented multi-signature account approach is that transactions submitted through it must be manually encoded by the client. This not only adds complexity on the client side but also means that a client could technically submit any type of transaction, even if it's not valid. From a security perspective, this shouldn't be a major problem since all operations in the system have authorization mechanisms and access controls in place, and invalid transactions would simply be rejected. For this latter reason, and since the presented multisig pattern is fairly standardized within Ethereum development, we concluded it was the best choice. Even so, other design alternatives were considered that aim to restrict operations callable by the multi-signature accounts.

Transaction Factory - An alternative would be to implement a *TransactionFactory* contract that included methods to create transaction proposals. When executing them, a switch-case would be used to call the appropriate method directly. The downside of this approach is that the *TransactionFactory* would need to comprise an extensive list of hardcoded endpoints (one for each operation), which is not great in terms of scalability and maintenance. Additionally, it would be complex to decide how to store transaction proposals since each transaction has its unique parameters.

Command Pattern - In contrast to the factory approach, we could try to separate the several operations between different smart contracts, creating something akin to a Command Pattern of object-oriented programming. These contracts would implement a *Command* interface with an *execute* method, which would invoke the desired contract's interface. However, complexity increases when we consider that depending on the command, the *execute* method would require the specification of different arguments, either through a state setter or through the *execute* function itself. Additionally, other concerns would arise regarding determining the command's original sender in the actual operation function. Since the *execute* method would be executed from within the *Command* contract, the transaction's context would be changed to that of the *Command* contract itself, making it the transaction sender. This would imply introducing additional traceability logic and authorization checks to know the original sender. The same issue goes for the transaction factory hypothesis.

Ultimately, these approaches are possibly feasible and could effectively simplify client interactions by reducing the steps required to execute commands. However they come at

the cost of introducing greater complexity at the smart contract level. Ultimately, we decided against adopting these approaches because we considered the increased complexity unjustified and felt it would make the solution less scalable, especially when introducing new functionalities into the system.

Another important consideration is that for the same multisig account, different types of policies might be desired for different operations. Therefore, the policy for transaction confirmation is not defined within the multisig contract itself but rather in the target contracts. In the proposed solution, contracts whose functions can be called by multisig accounts must implement a *Multisignable* interface. This interface includes a *getMultisigPolicy* operation, which specifies the policy to be applied for that particular contract. In the case of the *SelfMultisig* contract, it is itself *Multisignable*. This method also effectively limits the operations that can be executed by multisig accounts, which is an advantage given the considerations mentioned earlier.

5.4 Network Permissioning

The role of permissions is to define system access rules and identify the entities participating in it. The permissioning model we will consider in the scope of this project has already been described in section 4.3.
In Besu, onchain permissioning is implemented at the smart contracts level. This means that all logic associated with access control is fully customizable according to each project's requirements. To integrate permission smart contracts into an Hyperledger Besu network, it is necessary to implement two main smart contracts: one responsible for filtering client-submitted transactions and another for filtering node connections. In our solution, these contracts were named *AccountPermissions* and *NodePermissions*, respectively.

```
1  interface AccountPermissions {
2      function transactionAllowed(address _sender, address _target, uint _value,
           uint _gasPrice, uint _gasLimit, bytes calldata _payload) external view
           returns (bool);
3      function boot(address _cns) external;
4  }
5
6  interface NodePermissions {
7      function connectionAllowed(string calldata _enodeId, string calldata _ip,
           uint16 _port) external view returns (bool);
8      function boot(address _cns) external;
9  }
```

Figure 5.5: Account and Node Permission Contract's Interfaces

The functions *transactionAllowed* and *connectionAllowed* have predefined headers that these contracts must implement for Besu to recognize them as permission smart contracts.

Within these two functions, we can code three simple rules that form the basis of network permissioning:

- Which nodes may join the network;
- Which accounts/addresses may submit transactions;
- Which accounts/addresses may create smart contracts.

The implementation of these rules will be revealed later, but for now, it is more important to explain the remaining agents of the permissioning model. These include the organizations participating in the system, associated nodes and accounts, as well as their respective roles and permissions. There are four independent registry smart contracts that hold this information: the *OrganizationRegistry*, the *NodeRegistry*, the *AccountRegistry*, and the *RoleRegistry*.

5.4.1 Organization Registry

The *OrganizationRegistry* smart contract defines which organizations participate in the network and stores metadata associated with each. It includes functions to add new organizations and activate/deactivate existing organizations. Figure 5.6 shows the endpoints provided by this contract. For the following registry contracts, the interfaces will be hidden since their operations are similar.

```
1 interface OrganizationRegistry {
2     function addOrg(string calldata _orgId) external;
3     function orgExists(string calldata _orgId) external view returns (bool);
4     function isActive(string calldata _orgId) external view returns (bool);
5     function deactivateOrg(string calldata _orgId) external;
6     function reactivateOrg(string calldata _orgId) external;
7 }
```

Figure 5.6: Organization Registry Contract Interface

Questions may arise regarding the independent existence of the functions *addOrg* and *addParticipant* from the *OrganizationVoter*. The list of participants in the organizational multi-signature account is independent of the general list of organizations. This was chosen by design since there may be interest in adding organizations to the system that do not have voting power over critical operations. An example could be the financial institutions or banks, which could join the system in large numbers. This could potentially allow public or governmental institutions, which are naturally fewer in number, to be easily overwhelmed by their votes.

5.4.2 Role Registry

Each account registered in the system shall be assigned to a role. The *RoleRegistry* smart contract is the registry that stores the existing roles in the system and identifies the permissions associated with each. These permissions include permission to create accounts, roles, nodes, to deploy smart contracts, and even access to specific business workflows. An account can only create new roles with permissions that the creator account's role itself possesses. This premise effectively limits the permissions that an organization can have overall. For instance, when a new organization is introduced, it is required to define the address for the root admin of that organization and determine its permissions. If this root admin is not granted the permission to register nodes, then no member of that organization will be able to register nodes, rendering the organization node-less. This allows for the distribution of administrative powers between organizations.

The permission to deploy smart contracts is perhaps an account's most critical permission. As discussed in subsection 4.2.1, there are several implications in allowing an account to deploy smart contracts on a permissioned network without transaction fees. Therefore, this permission will be naturally granted to a very limited group of organizations, if any at all. Another approach is to restrict contract deployment strictly to the organizational multi-signature account (*OrganizationVoter*), which serves as an additional safeguard by requiring consensus before any deployment can occur.

5.4.3 Account Registry

The *AccountRegistry* smart contract is the main registry for accounts, maintaining track of every property associated with each. Each account is identified by an address, falls under an organization and has a role. It may also be active or inactive. Admin accounts can vote for the organization in the context of organizational multisig transaction requests.

5.4.4 Node Registry

The *NodeRegistry* smart contract is responsible for storing data related to network nodes. Each node is uniquely identified by a *enodeID*, and is associated with an IP address and a port. They also must be registered under some organization and may be deactivated and reactivated.

5.4.5 Permission Endpoints Smart Contract

The previously discussed contracts comprise the most critical operations within the system, as they fundamentally let us determine the network's participants, the permissions each one holds, and who is responsible for maintaining the infrastructure. For this reason, it is crucial to implement robust security and integrity checks within these contracts and access control rules following a structured governance model that aligns with the organizational

objectives and compliance requirements.
However, those smart contracts were designed to be decoupled from each other, in order to enhance the modularity of the solution. For this reason, the smart contracts do not communicate directly among them, thereby lacking the capability to perform all necessary integrity checks and authorization validations for their operations. To address this issue in the simplest and most untangled manner, a higher-level smart contract was developed to interact with the registry contracts and enforce these security rules. This is the purpose of the *PermissionEndpoints* smart contract. The critical operations specified in the registry contracts have their access restricted to the *PermissionEndpoints* contract (meaning only the latter can invoke them), which in turn makes them available through its own endpoints:

```
1  interface PermissionEndpoints {
2      function addOrganization(string calldata _orgId, address _admin,
           RoleRegistry.Permission[] _perms) external;
3      function addRole(string calldata _roleName, RoleRegistry.Permission[]
           _perms) external;
4      function addAccount(address _account, string _role, bool _isAdmin) external;
5      function changeRoleOf(address _account, string _role) external;
6      function deactivateOrganization(string _orgId) external;
7      function reactivateOrganization(string _orgId) external;
8      function deactivateAccount(address _account) external;
9      function reactivateAccount(address _account) external;
10     function addNode(string _enodeId, string _ip, uint16 _port, uint16
           _raftPort) external;
11     function deactivateNode(string _enodeId) external;
12     function reactivateNode(string _enodeId) external;
13 }
```

Figure 5.7: Permission Endpoints Contract Interface

addOrganization - To add an organization, it must be specified a new account to be assigned as its admin and the permissions that account will have (and subsequently, the rest of the organization). This account will be registered under a new role named *<org>_admin*.

deactivateOrganization/activateOrganization - These operations may only be invoked through the consensus of the rest of the organizations (through the *OrganizationVoter*).

addRole - This operation registers a new role under the invoker's organization. It is restricted to accounts with the permission to create roles. The registered role cannot have any permission that the one registering doesn't.

addAccount - This operation registers a new account under the invoker's organization. It is restricted to accounts with the permission to create accounts. The specified role must belong to the invoker's organization and the registered account can only be named admin if the invoker is also an admin.

changeRoleOf - This operation changes the role of an account. It is restricted to accounts with the permission to create accounts. The specified account and role must belong to the invoker's organization.

deactivateAccount/reactivateAccount - These operations are restricted to accounts with the permission to create accounts. The specified account must belong to the invoker's organization.

addNode - This operation registers a new node under the invoker's organization. It is restricted to accounts with the permission to create nodes.

deactivateNode/reactivateNode - These operations are restricted to accounts with the permission to create nodes. The node's organization must be the same as the invoker's.

Now that we have a grasp on how the permission model is implemented, let's jump back to the *AccountPermissions* and *NodePermissions* smart contracts. These are the two smart contracts that act as entry points to implement permissioning logic in Hyperledger Besu. They are responsible for filtering transactions submitted by clients (through the *allowedTransactions* endpoint) and for filtering node connections (through the *allowedConnections* endpoint), respectively.

5.4.6 Allowed Transactions

For a transaction to be accepted in the system, the sender's address must either belong to a smart contract that is part of the system and registered in the *ContractNameService*, or to an active account registered in the *AccountRegistry*. Additionally, the organization to which the account belongs must also be active. Regarding the creation of smart contracts, which is a type of transaction identifiable by not having a destination address, specific rules apply.

5.4.7 Allowed Connections

For a node to connect to the network, it must be previously registered in the *NodeRegistry* by an authorized member of the corresponding organization. The IP and port used by the node must also be predefined and specified in the *NodeRegistry* for security reasons. Any inconsistency in these configurations will prevent the node from connecting to the network, even if its *enodeID* is registered. This ensures that only verified and authorized nodes can participate, maintaining the integrity and security of the network.

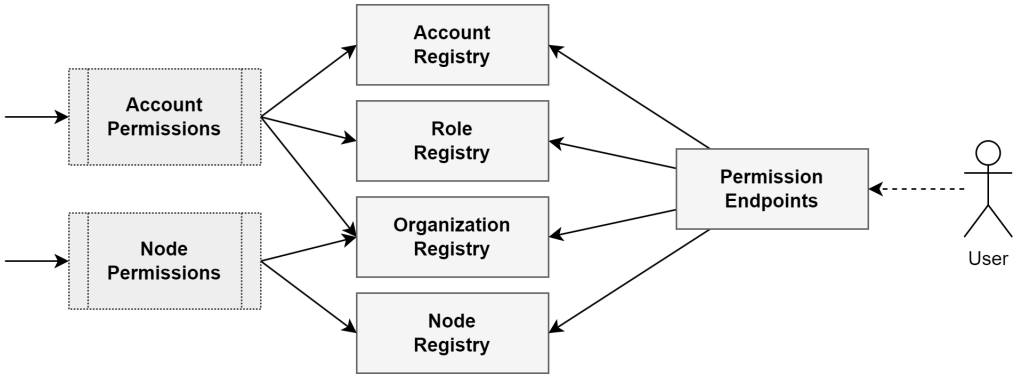


Figure 5.8: Permissioning Smart Contract Architecture Diagram

5.5 System Currency

Monetary transactions are an intrinsic component of the solution. Within a blockchain environment, several currencies may be created and supported through fungible token smart contracts. Public blockchains have the additional requirement of having a native cryptocurrency since here is a need to incentivize people to participate in the network. But in permissioned blockchain solutions, this is neither a requirement nor a standard. Although Hyperledger Besu is primarily designed for enterprise permissioned networks, it still supports the use of Ether (ETH) and allows the implementation of transaction fees and gas costs. This flexibility enables us to use ETH as the system's currency. However, to maintain a high level of generality in our solution and to ensure easy replication across other private blockchain frameworks that do not support native currency, we opted to use a regular fungible ERC20 token.

Token standards like ERC20 on Ethereum are essential for defining a common list of rules that all tokens must follow, providing a predictable interface for 3rd parties to interact with tokens across the Ethereum ecosystem. Nevertheless, it is still valuable to adapt these standards to private solutions. They offer a minimal yet rich list of endpoints that are secure and versatile, which should allow for the execution of nearly any functionality that might be desired. These standards bring scalability and modularity to the solution, as they are completely independent from other components.

```
interface IERC20 {
    event Transfer(address _from, address _to, uint value);
    event Approval(address _owner, address _spender, uint value);
    function decimals() external view returns (uint8);
    function mint(address _to, uint _amount) external returns (bool);
    function balanceOf(address _account) external view returns (uint);
    function transfer(address _to, uint _amount) external returns (bool);
    function allowance(address _owner, address _spender) external view returns
        (uint);
    function approve(address _spender, uint _amount) external returns (bool);
    function transferFrom(address _from, address _to, uint _amount) external
        returns (bool);
}
```

Figure 5.9: ERC20 Wallet Contract Interface

mint - Mint a certain token amount to the specified address. Only entities and organizations with the permission to mint currency have authorization to invoke this operation.

totalSupply - Returns the total token supply.

balanceOf - Provides the number of tokens held by a given address.

transfer - Allows a token holder to transfer tokens to another address.

approve - This function is used by a token owner to approve another address to move a specific amount of tokens from their account.

transferFrom - This function is used by an approved address in order to transfer tokens from one address to another, using based on the allowance given.

allowance - Returns the amount of tokens that an address is still allowed to move from another address.

For other contracts to be able to move currency between entities and properly function as escrows, it is crucial that the token owner first approves the contract to manage their funds. This is achieved through the *approve* function, where the token owner grants an allowance to the specified address, in this case, the contract address. This allowance specifies how much money the contract can move on the owner's behalf, ensuring that transactions are executed within the agreed-upon limits.

5.6 Property Documentation and Compliance

As discussed in chapter 3, most real world applications of digital assets use blockchain as a supporting infrastructure with the goal of ensuring integrity over some process.

Documentation associated with the property is usually emitted and stored entirely off-chain (in legacy systems), but their hashes are recorded within the tokenized property. However, if a blockchain network were to become the official infrastructure for issuing and storing these documents, it would be logical to also digitize and store them entirely within it. For each certificate and necessary documentation, the mission would involve having a smart contract on the chain that could replace traditional methods. Issuance would be fully on-chain. This issuance would still need to be performed by the entities currently responsible for these documents, acting in a role akin to an oracle - they would be tasked with verifying real-world information and inputting it into the blockchain via smart contracts. For instance, property appraisers could enter evaluations of the house worth between a certain range, energy companies might input values representing energy certificates, among other types of evaluations. With the development of these digital contracts, the system could be further enhanced. We could augment the functionalities with more stringent requirements and necessary documentation prior to transaction processes, and even implement entirely new features. This approach would allow many of the validations currently performed off-chain to be digitized, adding robustness to the system and ensuring both compliance and efficiency in automated real estate transactions.

However, the digitization of these documents is a long and time-consuming process. Given the nature of this thesis, it is neither pertinent nor productive to build a smart contract for every possible document. Instead, we present two smart contracts designed to exemplify the described approach. The first smart contract, *Compliance*, maps property types and operation types to the necessary documentation required for the execution of those operations. For example, this smart contract specifies that to sell a property of type X, documentation Y and Z are required. The second smart contract, *Document*, represents a digitized document. The existence of these smart contracts serves to illustrate how documentation would be integrated into the system, as the primary focus of this work is actually the representation of ownership records, which is discussed in the next section.

```
interface Compliance {
    event NewDocumentation(string indexed _target, string _service, address
        indexed _document);
    function documentation(string _target, string _service) external view
        returns (address[]);
    function addDocumentation(string _target, string _service, address
        _document) external;
    function removeDocumentation(string _target, string _service, address
        _document) external;
    function isCompliant(address _realty, string _service) external view returns
        (bool);
}

interface IDocument {
    event Issuance(address indexed _target, string _name);
    function name() external pure returns (string memory);
    function issuer() external pure returns (string memory);
    function expirationDate(address _asset) external view returns (uint);
}
```

Figure 5.10: Compliance and Document Contract Interfaces

5.7 Property Tokens and Fractional Ownership

As seen in section 3.2, creating assets that support fractional ownership in a blockchain environment, and specially Ethereum, is not very usual and thus can be quite challenging. This is because the fundamental idea of NFTs is to create a single entity on the blockchain, with unique characteristics and with a well-defined owner. They are by nature indivisible, meaning that a single token cannot be split and can only belong to a single owner. Even so, various authors proposed a few ways that aim to circumvent this issue by using other kinds of token standards.

An initial thought may be to use fungible tokens such as the ERC-20 and the ERC-777 token standards to represent ownership shares of an asset. For each real estate property, we could create a new token type and associate it with the metadata of the property it represents. Then, ownership of that property would be represented by a fixed number of tokens, with each token representing an equal share of that property.
Similarly, another idea involves using the ERC-1155 multi-token standard that can mint and store both fungible and non-fungible tokens. Within its implementation, there is a mapping of mappings that allows us to define a user's balance for a specific token type. If each token type was associated with the metadata of some property, then the balance a user has on that token type could represent his shares on the property, effectively allowing fractional ownership.
From our research, other approaches could even be explored involving token standards such as the ERC-404 for hybrid tokens and the ERC-998, which extends the ERC-721 with

the ability for NFTs to own other tokens.

However, there's a common issue that arises in all of these approaches involving token standards. They sure allow us to represent shares of ownership through fungible units, that can be recursively split and interchanged. Owners would be able to sell and transfer shares of a property, but there's not much more to it. It is not intuitive how the actual management of the token is supposed to be done. If the tokens were to represent something complex, like a real estate property, there would be several actions involving the token that may, for instance, require approval from every owner or require something resembling a voting system. Building this on top of these well-defined token standards would be complex and antithetical to their design. This is where multi-signature accounts will first come in handy.

Each NFT has, by nature, a unique owner, which in Ethereum terms may be the address of either an account or another contract. This alternative involves creating a multi-signature account for each property NFT and assigning the multisig's address as the NFT owner. By doing this, we are effectively allowing multiple people to own the asset through the multi-signature account. According to what was discussed in section 5.3, since multi-signature accounts allow for any transaction to be submitted, any action involving the property can be democratically controlled and executed. This approach makes up for a clean and versatile solution, offering great scalability and maintainability. Thus, our solution follows this design.

5.7.1 Property Ownership Implementation

Our implementation consists of two main smart contracts: *RealtyFactory* and *Ownership*.

```
1 interface RealtyFactory {
2     function mint(RealtyDetails _details, address[] _owners, uint[] _shares)
          external returns (address);
3     function getRealtiesOf(address _user) external view returns (address[]);
4     function detailsOf(address _assetId) external view returns (RealtyDetails)
5     function kindOf(address _assetId) external view returns (string)
6     function addOwnership(address _assetId, address _user) external;
7     function removeOwnership(address _assetId, address _user) external;
8 }
```

Figure 5.11: Realty Factory Contract Interface

RealtyFactory is the smart contract which is capable of minting and storing the property metadata. It also (and mostly) functions as a factory for *Ownership* smart contracts, which are the ownership multisig accounts associated with the assets. In the previous context, *RealtyFactory* would be the property NFT smart contract. The interface for the *RealtyFactory*

smart contract does not look like a NFT interface, however. This is because the transfer and transfer approval operations and logic are instead contained within the *Ownership* contract.

```
interface Ownership {
    function approvedOf(address _addr) external view returns (address);
    function approve(address _addr) external;
    function transferShares(address _from, address _to, uint _amount) external;
    function setAdmin(address _admin) external;
}
```

Figure 5.12: Ownership Contract Interface

The *Ownership* smart contract is a multi-signature account that represents shared ownership over a single asset. It is an implementation of the *WeightedMultisig* contract (subsection 5.3.1), and it applies the multi-signature logic to ownership shares. It limits the *WeightedMultisig* account to have 10.000 shares (meaning the least possible share is 0.01%) and complements it with new transfer approval mechanisms and the possibility to elect an admin, an account with permission to bypass the multi-signature requirements on operations that allow it.

One thing to note is that we assumed every owner's shares are sold independently, and therefore property tokens can always belong to the multi-signature account initially assigned to them. Thus, there's no need to have *transfer* and *approve* operations on the NFT smart contract itself (the *RealtyFactory*), as these are already present in the *Ownership* smart contract. But if it was desired to support transferring the ownership of an asset to a completely new set of owners within a single transaction, then it would be justified to keep the transfer and approve logic within the *RealtyFactory* smart contract. In that case, this operation would require the approval of all previous owners.

5.8 Sale Agreements

The most central functionality is the ability to create and manage sale agreements digitally through smart contracts. The goal is to streamline the sale process by automating the enforcement of contract terms and handling of the assets, making transactions more secure and efficient.

The *SaleAgreement* smart contract is the contract responsible for executing the transaction of goods within the context of sale. It operates as an escrow during the asset exchange process. For each sale, a new *SaleAgreement* contract is instantiated and initialized by providing the specific sale terms. These include the sale price, the earnest amount, the shares being transferred, contingency clauses and the real estate agent associated with the sale and respective commission, if applicable.

Before diving into the specifics of the actual implementation, it is important to revisit the core steps of real estate transactions, as was described in Table 2.1.
The general workflow of a real estate transaction typically involves multiple stages, starting with the visitation and negotiation phase. During this initial stage, the buyer and seller meet to discuss the property and negotiate terms. Once they reach a consensus on the terms of the sale, a pre-sale agreement contract is drafted, specifying the agreed-upon sale terms and including any contingency clauses. Following this, both parties are committed to proceeding with the transaction, unless any of the contingency clauses are invoked during the evaluation and inspection period. If a contingency clause is triggered, the contract may be withdrawn and loses its legal effectiveness. If all conditions are met, the parties proceed to sign the house deed, and the property is officially transferred.

The purpose of the SaleAgreement contract is to act as a substitute for the two main legal documents signed by both parties during the sale process: the **sale pre-agreement** and the **sale deed**. Thus, it will logically have a two-step nature.
When a *SaleAgreement* smart contract is instantiated, this action is equivalent to drafting the pre-sale agreement. The contract terms are specified through parameters, and even contingency clauses that are not enforceable by code are still recorded in the contract. Then, the contract offers the following interface:

```
1 interface SaleAgreement {
2     event PreSaleAgreement(address buyer, address seller, address indexed
          realty);
3     event DeedTransfer(address buyer, address seller, address indexed realty);
4     event SaleWithdrawal(address buyer, address seller, address indexed realty);
5     function consent() external;
6     function commit() external;
7     function widthdraw() external;
8 }
```

Figure 5.13: Sale Agreement Contract Interface

consent - This is the first function that must be called within this contract. As a requirement, both parties must first approve the contract (in the *Wallet* contract) to move the transacting goods for them. The *consent* action is equivalent to signing the terms of the pre-sale agreement contract. Thus, if the function executes successfully, the *SaleAgreement* contract will hold in escrow the property from the seller, and the earnest from the buyer. This way the transacting goods are effectively locked and cannot be used for anything else.

commit - The *commit* operation is the equivalent to signing the sale deed. Thus, it must be invoked by both parties following the property evaluation and inspection period, in case everything goes according to the plan and no contingencies are triggered.

The successful execution of this function performs the final transaction of the assets - the ownership of the property is transferred to the buyer and the entire sale price is deposited in the seller's account.

withdraw - The *withdraw* operation shall be invoked instead of the commit operation, in case one of the parties decides to back off from the deal and the contract must be reverted. This could happen if one of the contingency clauses is activated, allowing the affected party to back off with no legal consequences, or alternatively if a party decides do back-off with no valid justification. In the latter case, the party withdrawing would be breaching the contract, which would allow the other party to legally dispute this action and in most cases would require intervention of a legal authority. Penalties may be imposed on the party in breach of the contract. The *withdraw* function may be invoked by a privileged legal authority or, alternatively, by the consensus of both transacting parties.

5.8.1 Possible Workflows

Table 5.1 outlines the steps in the revamped workflow of a real estate sale process, illustrating how the described interface integrates into it. Depending on the specifics of the case, the latter steps may vary.
Steps highlighted in light grey are assumed to be handled entirely off-chain.

	Buyer	**Seller**
1.	Initiation through engagement of a real estate agent or lawyer	
2.	Mortgage loan due diligence (if applicable)	Sale legality verification and registration procedures
3.	Property visitation and negotiation of sale terms	
4.	Instantiation of new *SaleAgreement* reflecting the agreed-upon sale terms	
5.	Approval of the *SaleAgreement* to move the earnest amount	Approval of the *SaleAgreement* to move the property
6.	Invocation of the *consent* function, resulting in the transacting assets being held in escrow	
7.	Property inspection and evaluation period. Gathering of mandatory documentation by both parties	
8.	Securing mortgage financing. Approval of the *SaleAgreement* to move the remaining funds	

Table 5.1: Revamped Land Conveyance Process

The continuation of the workflow will depend on the outcomes of steps 7 and 8. Ideally, all conditions are met and the parties agree to proceed with the sale. The process will transition to the final stages of property handover as detailed in the subsequent steps of the table.

9.	Invocation of the *commit* function, resulting in the assets being successfully transferred between the parties
10.	Handover of the keys

If disagreements arise, parties may renegotiate and repeat the prior process. Alternatively, if one of the parties decides to back off, certain outcomes may follow. If both parties agree to withdraw the contract, the process may proceed without recurring to legal actions.

9.	Invocation of the *withdraw* function, resulting in the asset being transferred back to the seller and the earnest money returned to the buyer, except for any agreed-upon penalties

If the parties do not agree to withdraw the contract, a legal process may be initiated involving a judicial authority. Ultimately, this entity will determine whether the contract dissolution is justifiable. If justified, the contract is dissolved without penalties. If not, the party at fault may face a penalty. The penalization might include a fine deducted from the earnest held by the contract, or even being forced to proceed with the transaction against their wishes.

9.	Initiation of legal proceedings, evaluation of claims and determination of contract termination justifiability
10.	Invocation of the appropriate on-chain operation (*commit* or *withdraw*), depending on the outcome of the previous step

5.8.2 Design Considerations

As briefly mentioned in subsection 5.3.2, due to the multi-party signature nature of sale agreements, we decided to use a *SelfMultisig* in this smart contract. However, the multisig logic is implemented within the smart contract itself. In other words, users do not need to worry about encoding, submitting and confirming the transactions. They only need to invoke the desired function (*consent*, *commit*, or *withdraw*), and the confirmation logic for these operations is executed autonomously in the background. This approach greatly simplifies and facilitates interaction with the smart contract.

Another consideration when developing this smart contract was whether there is any advantage in generating a new *SaleAgreement* smart contract for each sale conducted, instead of merely storing the sale details in a structure and conducting the transaction within the factory contract themselves. Both alternatives can be valid depending on the environment, each with their pros and cons. The second alternative is the most common in public blockchain environments. This is because it is the most cost-efficient, by far. Deploying a new smart contract for each transaction is very gas-intensive when comparing to executing transactions within an existing contract. However, this alternative loads the

central contract with more complexity, increasing the risk of exploits. Additionally, as the number of transactions grows, the central contract could become a bottleneck, as it could grow indefinitely. But as mentioned before, gas efficiency is not such a deal-breaker concern in permissioned blockchain environments, as resource savings are less substantial. Thus, we decided to opt for the modular and isolated approach of having individual *SaleAgreement* contracts. This alternative also provides flexibility in case different kinds of contracts are eventually developed, and enhances upgradability and maintainability since the code of the contracts is independent from the code of the factory contracts.

This last consideration can also be done for the rental and loan smart contracts, which we will discuss next.

5.9 Rental Agreements

Rental or lease agreements are legal documents that contain clauses specifying the rental terms and conditions agreed upon between tenants and landlords for the use of property. These agreements are enforceable by law and outline responsibilities such as payment of rent, maintenance of the property, and other tenant and landlord obligations. While rental agreements generally do not need to be registered in an official database in most countries, they must be declared and are often subject to taxation. Rental accommodations can vary significantly, but for the purpose of this discussion, we will consider the most common and generic scenario: monthly tenancies.
The *RentalAgreement* is a smart contract designed to encapsulate the typical terms found in rental or lease agreements. This contract aims to represent a relatively simple and generic rental agreement without overly specific clauses. More specific contracts could potentially be introduced into the system in the future, even accommodating other types of lodgings such as local accommodation or motel-style accommodations.

Instantiation of a *RentalAgreement* contract for a property can only be done by its own *Ownership* contract through a dedicated factory contract. This means that a rental agreement for a given property can only be created if an agreed set of its owners agree to do so, according to the policy defined in the factory contract.
It implements a *PaymentSplitter* contract, a relatively simple and generic smart contract developed to implement payment division logic. In a *PaymentSplitter* contract, it is possible to define a group of payees and their respective shares. A payment sent to the *PaymentSplitter* through its *pay* function is divided among the payees in proportion to their respective shares. The list of payees defined in the *RentalAgreement* is intended to be independent from the property owners. We assume that the ownership percentage of an asset does not always reflect the payment division agreement. It is reasonable to assume that there may be scenarios where one of the landlords receives a larger share due to being responsible for tasks associated with property maintenance, for example.

By keeping these lists independent, we also open possibilities for third parties to receive direct on-chain payments for services that may be associated with the rental.
The PaymentSplitter provides the following interface:

```
interface PaymentSplitter {
    event PaymentReleased(address to, uint amount);
    event PaymentReceived(address from, uint amount);
    function getPayees() external view returns (address[] memory);
    function getTotalShares() external view returns (uint);
    function sharesOf(address _account) external view returns (uint);
    function pay(uint _amount) external;
    function collect() external returns (uint);
}
```

Figure 5.14: Payment Splitter Interface

getPayees - Retrieves a list of all payees registered in the contract along with their respective shares.

getTotalShares - Returns the total sum of shares distributed among all payees, providing a basis for calculating individual percentages.

sharesOf - Returns the number of shares owned by a specific payee, indicating their proportion of the total payment.

pay - This function allows the invoker to make a direct payment, which is then split among the payees according to their shares. This function requires that the payer approves the contract with sufficient allowance to make the payment.

collect - Distributes any funds held by the contract among the payees according to their shares. This function allows payments to be made through the *Wallet* contract, thus not requiring prior approval.

The *RentalAgreement* contract extends previous functionalities and adapts them to the rental context.

```
1  interface RentalAgreement {
2      event RentalEnrolled(address tenant, address landlord, address indexed
           realty);
3      event RentalComplete(address tenant, address landlord, address indexed
           realty);
4      event RentalTerminated(address tenant, address landlord, address indexed
           realty);
5      event TermRenewed(address tenant, address landlord, address indexed realty);
6      event TermReduced(address tenant, address landlord, address indexed realty);
7      event RentPayment(address tenant, address landlord, address indexed realty,
           uint value);
8      function enroll() external;
9      function payRent() external;
10     function terminate() external;
11     function evict() external;
12     function returnDeposit(uint _penalty) external returns (uint);
13     function reduceTerm(uint _periods) external;
14     function renewTerm(uint _periods) external;
15     function paymentExpiration() external returns (uint)
16     function paymentDueDate() external returns (uint)
17     function getTerms() external view returns (RentalTerms memory);
18 }
```

Figure 5.15: Rental Agreement Contract Interface

enroll - The *RentalAgreement* contract is activated once it is enrolled by the tenant, reflecting their agreement to the terms specified in the contract. This operation will transfer the security deposit and the first month of rent from the tenant's account to the *RentalAgreement* contract and the landlord(s), respectively.

payRent - The function the tenant must use to pay the monthly rent. Underneath it uses the *pay* function provided by the *PaymentSplitter* to correctly distribute the rent between the determined payees.

terminate - This function is used to immediately terminate the contract of the tenant if both parties agree to do so (through multi-signature logic), or if a judicial authority intervenes.

evict - The landlord(s) may evict the tenant, if they fail to pay rent on the due date. This action immediately terminates the contract, and distributes the security deposit between the landlords.

returnDeposit - The deposit may be returned after the termination of the contract. A penalty may be specified to compensate for any damages. If enough time passes and the landlord has not yet returned the security deposit, the tenant itself may claim it without penalties.

reduceTerm - This operation allows the tenant to pay an early termination fee to shorten the rental term, provided the new termination date complies with the early termination notice defined in the contract terms.

renewTerm - Once the rental term has been concluded, its duration may be extended if both parties agree to do so (through multi-signature logic).

paymentExpiration - Calculates the day until the last payment is due.

paymentDueDate - Calculates the next payment due date.

getTerms - Retrieves the terms of the contract, including start date, rental duration, monthly payment amount, and early termination policies.

5.9.1 Implementation of Clauses

To better understand how these smart contracts can be applied to the real-world scenarios and workflows, let's review the most common clauses typically found in lease agreements and discuss their implementation within the *RentalAgreement* smart contract. We focused on clauses that are implementable by code, but it should be noted that there are some clauses that cannot be enforced without recourse to some form of off-chain oracle. In such cases, it would be most reasonable to assume one of two outcomes: either there is goodwill from both parties leading to a mutual termination of the contract through the contract endpoints, or alternatively, that there is a judicial entity on-chain with authority over any rental contract who can, upon investigating the situation, enforce the termination of the contract. Any penalties charged in addition to those concerning the security deposit are assumed to be handled off-chain as well, as they would likely be tied to a judicial process. The following clauses were gathered from an exemplary template of a generic lease agreement, referenced from [22].

1. Rent - The tenant is allowed to occupy the premises starting on (_) and ending on (_). The tenant shall pay the landlord(s), in equal monthly installments, $(_) of rent value. The rent shall be due on the (_)th of every month.
Implementation: The starting date and contract term duration are specified on initialization. The payment of the monthly rent installments is facilitated by the *payRent* function. This action must be manually completed by the tenant each month.

2. Security Deposit - The landlord(s) requires a payment of $(_) as Security Deposit, for the faithful performance of the tenant under the terms and conditions defined in the agreement. The Security Deposit shall be returned to the tenant within (_) days after the end of the lease term less any itemized deductions. Payment of the Security Deposit is usually required upon signing the agreement.
Implementation: The transfer of the security deposit is facilitated through the *enroll*

function. It is then returned though the *returnDeposit* operation. This function allows the landlord(s) to apply a penalty over the returned amount. If the determined amount of days passes and the security deposit has not been returned, the tenant can force this operation.

3. First Month's Rent - The Tenant is required to pay the first month's rent upfront. This can usually be done upon signing the agreement, or alternatively upon (_) days of the lease term.
Implementation: The first month's rent is immediately paid upon enrolling, along with the security deposit. The *enroll* operation will fail otherwise.

4. Late Fee - There shall be a penalty of $(_) every day the rent payment is late. Alternatively, this penalty could be a fixed value.
Implementation: The penalty amount is specifiable during the creation of the contract. In the *payRent* function, the amount due is increased by the specified penalty multiplied by the number of days late. There is no option to specify a fixed penalty value in the current implementation.

5. Early Termination - The tenant has the right to terminate the rental agreement at any time by providing at least (_) days of written notice to the landlord(s) along with an early termination fee of $(_). The tenant will remain responsible for the payment of rent during the notice period.
Implementation: Both the early payment fee and the termination notice are specified at the time of creating the rental agreement. The *reduceTerm* function utilizes these values. These values may also be set to zero if there are no restrictions regarding early termination. If the intent is to disallow early termination, the value of the early termination notice must be greater than or equal to the rental term.

6. Default - The tenant will be in default if he fails to comply with any of the terms agreed upon signing the contract, or if he fails to pay rent when due. The landlord has the right to terminate the agreement if that comes true.
Implementation: The contract can be forcibly terminated either through the *terminate* or the *evict* operations. If the tenant has failed to pay the rent at the due date, the *evict* function will immediately end the contract and transfer the security deposit to the landlord(s). For reasons other than payments, the *terminate* function allows the ending of the contract if both parties agree on doing so. In cases where the parties disagree, the matter should be reviewed by a judicial authority, which in turn will enforce the termination of the contract using the same *terminate* function if justified.

7. Term Renewal - The lease agreement may be renewed upon its completion. If a renewal is made, the tenant may continue to lease the premises under the same contract

terms.
Implementation: Renewal is allowed through the *renewTerm* function.

8. Sale of Property - If the property is conveyed to another party, the tenant shall be notified of the new owner. The new owner may or may not have the right to terminate the rental agreement by providing (_) days' notice.
Implementation: In the scenario where a rented property is sold to another party, the new owner does not have the authority to outright terminate the existing rental contract, though they are still allowed to shorten it as defined in the early termination clauses. Regarding the redirection of monthly payments, although the list of payees for the rent is technically independent from the property owners, it can be altered at any time by them. This means that if a new individual takes possession of the asset, they must use the *Ownership* contract to submit an operation that updates the payees in the rental contract.

5.9.2 Clauses not enforceable by code

Many clauses that are commonly specified in rental agreements are not verifiable or enforceable by code. Examples of these include policies related to safety, smoking, pets, noise, waste, etc. Even so, it is still necessary for them to be specified in some form in the *RentalAgreement* smart contract, as they exist and have legal value, requiring the tenant's signature. However, regardless of how they are specified in the contract, there is not an automatic way to verify compliance with these clauses that doesn't involve human intervention.
Hypothetically speaking, we could technically have an oracle system with smoke or noise sensors installed on the property that would allow the blockchain to automatically confirm compliance with these policies through an oracle system. But this scenario is neither realistic nor practical. Additionally, many clauses, such as those regarding noise, can even be subjective, complicating their verification even further. Therefore, the validation of these clauses shall be done in a similar manner to what currently happens: If a landlord suspects a breach of contract terms by the tenant, the landlord should address the tenant directly to resolve the issue. If the tenant acknowledges the violation, they may voluntarily vacate the property. If there is a dispute, it may require the involvement of a judicial authority to impartially evaluate the situation, potentially terminating the contract and imposing penalties on the party at fault. This approach is complicated and have additional costs to the parties, and therefore it is rare to happen.

The *RentalAgreement* smart contract can be terminated at any time if both parties agree. The *terminate* operation is expected to be used in these cases. If there is no consensus, a privileged authority may intervene and enforce this operation on the contract. Clauses that are not enforceable by code are assumed to be specified in the format of a byte array,

that can be translated into plain language and thus enforced through external means.

5.9.3 Rental Agreement Considerations

Similarly to the *SaleAgreement* smart contract, the multi-party signature nature of rental agreements led us to make it inherit *SelfMultisig's* characteristics. The multisig logic is implemented within the smart contract itself, so that users do not need to worry about encoding, submitting and confirming the transactions. They only need to invoke the desired multisig function (in this case, *terminate* or *renewTerm*).

The next logical step for the presented solution for rental agreements could be the integration of automated recurring payments. That is, instead of requiring the tenant to manually pay each month using the *pay* function, the rent amount could be automatically debited and transferred to the landlords' account. Opting for this design alternative would bring added complexity, to ensure a response in scenarios where the tenant does not have sufficient balance to make the automatic payment, for example. Nevertheless, it is still a good idea. In the case of Ethereum, however, it is not possible to make automated recurring payments on-chain. Ethereum does not have any native time-based transaction triggering mechanism. But this does not mean that this feature is not feasible. In a public blockchain network, this typically would involve recourse to an oracle. A use case for oracles, in this context, is to connect to an oracle that triggers transactions every month, signaling the network that a new month has begun. Then, the RentalAgreement contract, along with any other time-sensitive contracts, could be triggered in response to the oracle transaction. This approach can be replicated in various ways in permissioned networks. A very rudimentary and manual way to implement an oracle would be to have a *timeEvents* smart contract in which it's possible to call a function *monthPassed* every month, which would cascade all operations in the system that depend on this event. Within this function itself, it would be possible to verify if indeed a month has passed since the last call, as Solidity allows calculations with time. This manual approach can serve as a basic fallback mechanism where more sophisticated or external oracle services are not available or feasible.

5.10 Mortgage Loans

Mortgage loans are among the most common procedures when purchasing a home. Such is because it's rare for a buyer to have the capacity to purchase a property outright. There are various types of mortgage loans, which may vary from country to country, each with its own variables and clauses. We decided to proceed with an implementation of a fixed-rate mortgage loan. These are characterized by uniform interest rates and equal monthly payments throughout the life of the loan, which is a good fit for the demonstration purposes of this work. Before moving on to the details of the implementation, it is important to explain the mortgage concepts considered in this implementation.

Loan Principal - The original amount of money borrowed in the mortgage.

Down Payment - The upfront part of the purchase price paid by the buyer from their own funds.

Interest Rate - The percentage of the loan charged by the lender for borrowing its money.

Loan Term - The duration over which the loan is scheduled to be paid back.

Amortization - The process of paying off debt with a fixed repayment schedule in regular installments over time.

Grace Period - The period after the due date of a payment during which the borrower may still pay without penalty.

Default - Failure to meet the conditions of a loan, which may lead to the potential loss of the property to foreclosure.

Equity - The difference between the market value of a property and the amount still owed on the mortgage. In many cases this may also represent the fractional ownership between the borrower and the lender.

Fixed-rate mortgage loans were integrated into the solution through a single contract, the *FRLoan*. The most important detail to mention first about this contract is that it is another implementation of the multi-signature account pattern. The fundamental idea behind the *FRLoan* is to function as a separate entity, that must be charged with a certain amount of money, and which is capable of performing any type of transaction with it, given the approvals from the borrower and the lender. This means that this contract allows loans for anything within the system, and is not necessarily restricted to the purchase of a property.
If used for a property purchase, then a purchase transaction must be submitted and confirmed by both parties. Then, if the purchase is successful, the ownership of the asset will be transferred to the mortgage contract, instead of going directly to the buyer. Since the mortgage contract is a multi-signature account owned by both the lender and the

borrower, they are indirectly the owners of the asset. Any action concerning the asset will require approval from both parties until the loan is fully paid off. The fraction that each party owns of the multi-signature account is proportional to the portion of the loan that has already been paid off. The *FRLoan* contract encapsulates all the logic relative to the amortization and termination of the loan.

```
1  interface FRLoan {
2      event LoanEnrolled(address indexed lender, address borrower);
3      event LoanSecured(address indexed lender, address borrower);
4      event LoanAmortized(address indexed lender, address borrower);
5      event LoanTerminated(address indexed lender, address borrower);
6      event LoanForeclosed(address indexed lender, address borrower);
7      function enroll() external;
8      function secure() external;
9      function amortization() external view returns (uint);
10     function amortize() external;
11     function applyPenalty() external;
12     function foreclosure() external ;
13 }
```

Figure 5.16: Fixed-Rate Loan Contract Interface

enroll - Initialization function, that must be invoked by the borrower. This operation attempts to transfer the down payment amount from the borrower's account to the contract itself. Upon initialization, the borrower becomes the sole participant of the multisig account, thus having the right to use the down payment for anything he opts to.

secure - The next step is for the lender to finance the principal. This operation transfers the principal amount from the lender's account to the contract itself. Proportional multisig participation shares are given to the lender. After this, the mortgage loan becomes active and any operation regarding the principal amount will require consensus between the lender and the borrower.

amortization - Returns the amount that the borrower must pay periodically throughout the loan term.

amortize - The function to pay the amortization value. This operation attempts to transfer the amortization value from the borrower's account to the contract, and balances the equity afterwards. The good ending for a mortgage loan is to be terminated after fully paid. Thus, the contract is automatically terminated when the last amortization payment is done. By the end of the amortization schedule, the borrower will once again be the single multisig participant of the mortgage loan contract, which allows him to finally take possession of the asset.

applyPenalty - If the grace period for a payment has already passed, the lender may increase the interest rate according to the penalty specified at the mortgage terms.

foreclosure - If the default deadline for the payments has passed, the lender may foreclose the property. The *foreclose* action simply removes every single multisig share the borrower has over the loan contract. This allows the lender to take possession of the asset for itself in compensation for the rest of the debt owed.

The amortization value is calculated according to the well known monthly payment formula for fixed-rate mortgage loans, given by:

$$M = P \times \frac{r(1+r)^n}{(1+r)^n - 1}$$

Where:

- **P** is the loan principal.
- **r** is the monthly interest rate (annual interest rate divided by 12).
- **n** is the total number of payments (loan term in years multiplied by 12).

A relevant aspect in Ethereum development (and not only) is that Solidity does not support floating point types. All calculations must be performed using integer types, which introduces additional complexity in handling decimal values. To manage this, values are typically scaled by a factor to simulate decimals. In our implementation, we considered 2 decimals, and therefore the minimum monthly interest rate that is admitted is 1% (0.01).

Finally, it is important to discuss how the *FRLoan* smart contract would be integrated into the actual loan approval and sale process, as there are a series of stages and off-chain protocols that must be carried out before a loan is actually granted. First, let's see how the workflow of a successful sale (using a *SaleAgreement*) changes when the buyer takes out a mortgage.

	Lender	Borrower (Buyer)	Seller
1.	Definition and Review of Mortgage Contractual Terms and Conditions		Sale legality verification and registration procedures
2.	Mortgage loan due diligence		
3.	Pre-approval of the Home Purchase and Instantiation of new *FRLoan* smart contract	*FRLoan* enrollment	
4.	Initiation through engagement of a real estate agent or lawyer		
5.	Property visitation and negotiation of sale terms		
6.	Instantiation of new *SaleAgreement* reflecting the agreed-upon sale terms		
7.		Approval of the *SaleAgreement* to move the earnest amount	Approval of the *SaleAgreement* to move the property
8.		Invocation of the *consent* function of *SaleAgreement*, resulting in property shares and earnest amount being held in escrow	
9.	Property inspection and evaluation phase		
10.	Secure mortgage financing (Invocation of *secure*)		
11.	Approval of the *SaleAgreement* to move the principal amount		
12.	Invocation of the *commit* function of *SaleAgreement*, resulting in the assets being successfully transferred		
13.	Handover of the keys		

Table 5.2: Mortgage Loan Financing Workflow

The first phase, corresponding to the definition of the mortgage terms, is assumed to be fully performed by off-chain communication between the lender and the borrower. The terms of the loan will, however, later be reflected in the loan's smart contract. The due diligence phase corresponds to the analysis of a large set of factors and variables, mainly related to the financial stability of the borrowing entity. How this analysis is performed and whether any of it is done on-chain depends on how complete and comprehensive the (real-life) system is. If the totality or a large part of the information needed for this phase is stored on the chain, then the system could include verification mechanisms to automate the process. However, it is safer to assume that not all information will be available on the system, necessitating off-chain intervention.

After careful verification of all necessary data, if the lending entity agrees to provide a loan, then the pre-approval is reflected on the chain with the creation of the *FRLoan* smart contract. The existence and initialization of this smart contract indicates that the lender has accepted the possibility of providing a loan with the terms defined in the contract.

In the *FRLoan* smart contract, the borrower must *enroll* and submit a transaction for the

consent of a *SaleAgreement* smart contract, which will use the down payment as earnest. The lender will not finance the *FRLoan* smart contract until having properly evaluated the asset and decided that it meets the loan value. If he approves, then the mortgage will be successfully secured and a transaction *committing* the sale agreement may be submitted. This action marks the phase of credit approval for the home purchase. After both the lender and the borrower confirm this transaction, along with the sale being committed by the seller, the purchase is executed, automatically transferring the funds to the seller and the property to the buyer, which in this case, is the *FRLoan* contract.
The *FRLoan* smart contract will then be continuously used to conclude the mortgage through the amortization schedule. This may have multiple outcomes depending on the payment attendance.

Ideally, the borrower continues to make regular payments until the loan is fully amortized. For each payment, the borrower is rewarded with equity over the *FRLoan* contract. By the end of the payment schedule, the borrower will own 100% equity over the mortgage, meaning the lender will no longer have rights to submit and confirm transactions submitted on the mortgage contract. This means the borrower entity will be able to freely transfer the ownership of the asset to his own account.

	Lender	**Borrower**
14.	Penalty application in case payments are late	Continuous Payment of scheduled installments
15.	Loss of rights over the mortgage contract	Termination of the contract by transferring the property to the borrower's account

The other main possible outcome is the failure to meet payment obligations, in which case the lender has the rights to foreclose on the mortgage.

14.	Initiation of foreclosure proceedings	Failure to make scheduled payments
15.	Termination of the contract by repossessing the property	Termination of the contract by loss of property

The interaction workflow between the *FRLoan* and the *SaleAgreement* contracts are visually described by Figure 5.17.

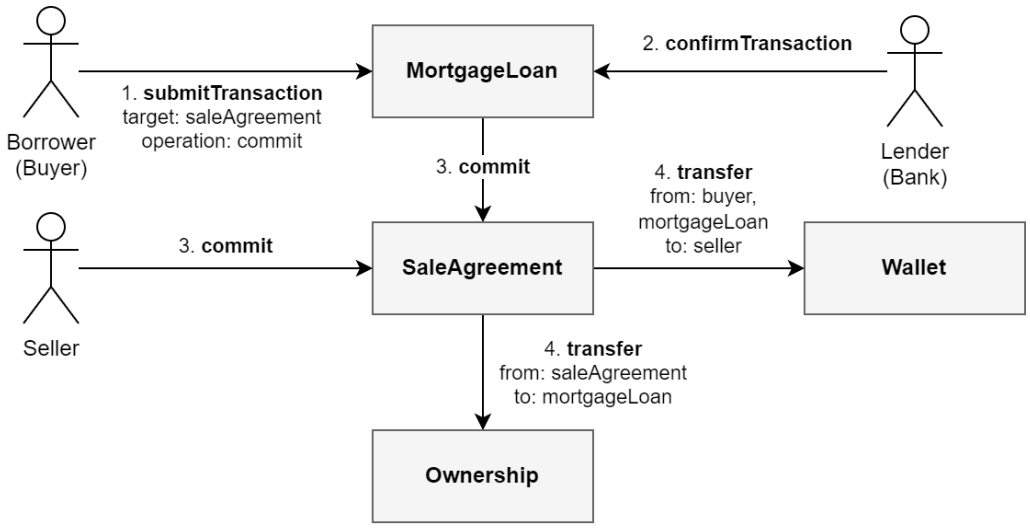


Figure 5.17: Mortgage Loan Flow Diagram

One advantage of the presented implementation of loans is that the loan contract is completely decoupled from the sale and rental contracts, hence loans being allowed for anything within the system, other than the purchase of a property. The counterpart disadvantage is that the transfer process of the asset, upon full debt payment completion, shall be done manually by the borrower, instead of being performed autonomously by the contract itself.
Furthermore, being an implementation of a multi-signature wallet means that the contract contains a function that allows the parties to submit any kind of transaction in the name of the loan contract - *submitTransaction*. Without this mechanism, landlord-reserved actions over the property would be not accessible to a mortgaged property.

5.11 Web Client Application

To facilitate the demonstration of the proposed solution, a web client application was developed featuring a graphical interface that offers an enhanced user experience for interacting with the presented smart contracts. The application was developed in *React* with *TypeScript*, and uses the *ethers.js* library to interact with the network. In addition, we chose to use the *Metamask* wallet service to facilitate authentication and account management. The application integrates all the necessary functionalities for transferring, buying, selling, and leasing properties. Loan functionalities, on the other hand, have been left out due to their complexity. These functionalities are still accessible through the Command-line Interface (CLI).
Appendix C provides a detailed visual overview of the application's interface, showcasing the various developed pages and their integration. More information on how to set up the network and use the application developed to interact with it, as well as a demonstration video can be found in the official GitHub repository of the *Blockopoly* dissertation, available at *rmarede/Blockopoly*.

6

Evaluation

The proposed solution was evaluated regarding its correctness and performance. For the system's correctness, the evaluation focused on testing the functionality of individual smart contract functions. This was achieved through the implementation of unit and integration tests, ensuring that each function performs as expected and that they interact seamlessly with the other components of the system. For the performance evaluation, the system was be subjected to multiple tests with varying numbers of peers and client requests. During this process, we measured and monitored key performance indicators including latency - the time taken for transactions to be confirmed - and throughput - the number of transactions processed per time unit. These metrics are relevant in understanding how the system performs under different conditions and provide insights regarding its efficiency and scalability. Hyperledger Caliper is the blockchain benchmarking tool that we used for this purpose.

6.1 Unit and Integration Tests

Certainly, the most critical aspect to evaluate in the proposed solution is the correctness of the implemented functionalities. Due to the irreversible nature of smart contracts, once deployed, the code that will be executed cannot be altered. This means that if there are bugs or exploits in the solution, they cannot be fixed. Although there are some strategies to address this issue (discussed later in subsection 6.4.1), this is the general rule. Therefore, it is crucial to have the highest possible confidence in the security of the developed architecture. To assess our solution in terms of its correctness, we implemented unit and integration tests. Unit tests are designed to test individual components of the software to ensure each part functions correctly. Integration tests, on the other hand, verify that different components of the system work together as expected. It is important to clarify that pure integration testing typically involves a comprehensive test environment where the complete solution is deployed, testing interactions across all components at the system level. However, the integration tests conducted in this work aim to evaluate the

interaction between smart contracts dependencies on a limited test environment where only a subset of dependent contracts are deployed, and therefore they are best labeled as contract integration tests.

To implement both kinds of tests, we made use of Hardhat's testing framework, which allows to write test scripts using JavaScript. Hardhat is an Ethereum development environment that facilitates performing tasks such as running tests, compiling and deploying smart contracts. It integrates a local Ethereum network simulation that makes it possible to run the unit tests for the developed smart contracts without the need to deploy the entire network. Hardhat's testing framework includes tools that help us build fixtures, assert function call behavior and results, and manipulate the network's internal time, which is essential for testing the time-sensitive functionalities in contracts like the *RentalAgreement* and the *FRLoan*. However, one limitation of Hardhat is its lack of a built-in mocking tool. Mocking is essential for scalable testing because it allows developers to simulate and isolate behaviors in smart contracts independently from their dependencies.
In our solution, we strived to modularize the architecture of the smart contracts as much as possible. Nonetheless, without mocking, there were instances where it was not feasible to conduct truly unitary tests due to dependencies between contracts. While Smock is a prominent mocking library developed by Defi-Wonderland that complements Hardhat with mocking capabilities, it is only compatible with version 5 of Hardhat, which is outdated and doesn't provide many of the tools the later versions comprise. Consequently, we opted for the traditional method of mocking smart contracts by creating mock contracts. For each smart contract we wanted to mock, we developed a fake version that implements the interface of the original smart contract but does not contain internal logic. These mock smart contracts also include functions that allow developers to set return values or revert the execution of the functions provided in the interface of the original smart contract.
A combined total of 188 unit and contract integration tests were developed with Hardhat. Running them is as simple as executing a single command. The full list of tested scenarios can be consulted in Appendix B. More information on how to run the unit tests is available in the appropriate directory at the *Blockopoly* GitHub repository [3].

6.2 Performance Testing

After ensuring the system correctness, the solution should be put under real-world conditions supported by an infrastructure that can withstand the load users will generate when interacting with it. This phase mostly cares about the non-functional requirements, putting the application under stress and performance tests, validating security expectations, and guaranteeing that the system is viable for deployment.
Performance tests are significantly influenced by the network topology, network bandwidth and latency, and the characteristics of the nodes that make up the network. Replicating such conditions in a test environment is a very complex task, and thus it is challenging to predict how the network would react to the same tests in a production environment. In addition, the accessible laboratory test environment for our benchmark evaluation is largely limited by the hardware specifications of the benchmarking machine (Table 6.2). To gain a better understanding of Hyperledger Besu's performance potential, independent of this study's context, we recommend consulting prior research conducted for this purpose [25].

Component	**Specification**
Processor (CPU)	Intel Core i5-6300HQ @ 2.30GHz (4 Cores, 4 Logical)
Memory (RAM)	16 GB DDR4
Graphics Card (GPU)	NVIDIA GeForce GTX 950M
Storage	TOSHIBA MQ01ABD100 HDD
Operating System	Ubuntu 20.04.6 LTS (WSL2) on Windows 10 Pro (Host)

Table 6.1: Benchmarking Machine Specifications

Despite these limitations, it is still relevant to benchmark the proposed solution since it is a way to get insights on the performance of some of the system's workflows. To make the hardware limitations less impactful, we conducted a control test which performs a very simple memory operation on a dummy smart contract - updating a mapping record. The other tests simulate procedures we could imagine being more frequent in a real-world setting. The goal of this evaluation is to compare the performance of the proposed solution to the control test in order to understand the additional overhead it introduces.

The system was be subjected to multiple tests with varying **numbers of peers** and **client requests**. During this process, we measured and monitored key performance indicators including latency - the time taken for transactions to be confirmed - and throughput - the number of transactions processed per time unit. These metrics are relevant in understanding how the system performs under different conditions and provide insights regarding

its efficiency and scalability.

Hyperledger Caliper is our blockchain benchmarking tool. Testing in Caliper is conducted through the specification of workloads. Workloads are predefined sets of tasks that ideally represent common use cases of the developed application. For this phase, we developed workloads for the following two use cases:

1. **Transaction of a Property via a Sale Agreement:** This workload tests the functionality of the *SaleAgreement* smart contract. It involves all intermediate steps, such as instantiating of the contract, consenting and committing.

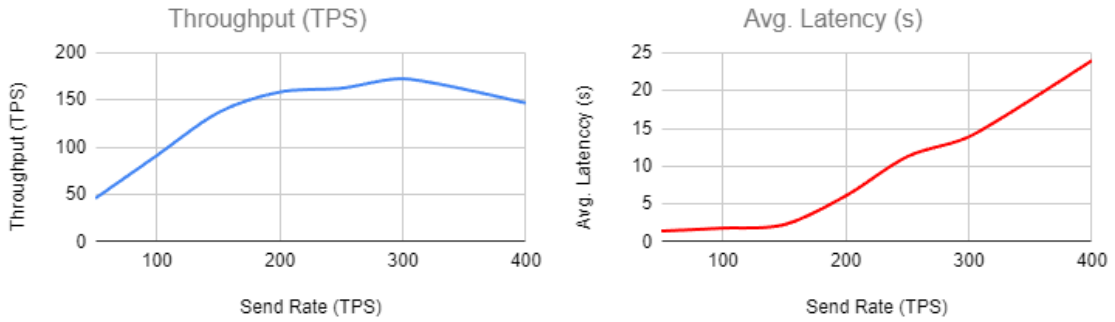
2. **Rental Use Case:** This workload covers a possible lifecycle of a rental agreement. It involves the instantiation of a *RentalAgreement* smart contract, executing the first monthly rent payment, and finally use the early termination feature of the contract.

All the tests were conducted on a network of 5 besu nodes, using the IBFT consensus protocol. Each Besu node runs in a docker container using the *hyperledger/besu:24.7.0* docker image. The benchmarking client is deployed on another docker container, this one using the image *hyperledger/caliper:0.6.0*. The benchmarking client also leverages a custom Caliper-Ethereum connector, specifically developed to allow the dynamic specification of contract addresses when sending requests. This is an important feature within our test environment since we want to benchmark the deployment of multiple sale and rental contracts, and interact dynamically with them.

6.2.1 Control Test

The control test allows us to estimate that in our environment, the maximum throughput for a simple operation on a smart contract is reached at a transaction send rate of around 300 TX/s (transactions per second), with a throughput of about 170 TX/s. For higher send rate values, it is expected that the throughput will decrease considerably. Regarding latency, it was observed that it starts to increase more significantly from around 150 TX/s.

These throughput values are very low in magnitude compared to what is expected for the performance of Hyperledger Besu in a proper production-ready environment, which is expected given the hardware limitations of our test environment. For reference, performance studies conducted by C. Fan et al. [25] reveal an average throughput of 400 TX/s for write operations for the same network size and consensus algorithm.



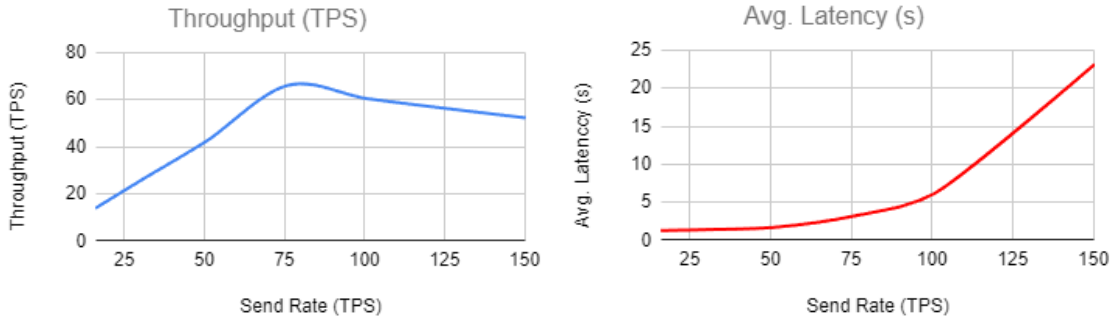
(a) Throughput Variation

(b) Latency Variation

Figure 6.1: Control Benchmark

6.2.2 Rental Use Case

The benchmark for the rental use case employs a mix of transactions to deploy new rental contracts and successively interact with them. In this case, the maximum throughput is reached at a transaction send rate of around 75 TX/s, with values nearing 66 TX/s. For higher send rate values, the throughput decreased considerably, stagnating at around 55 TX/s. As for latency, similar response times to the control test were observed for the lower transaction send rate range.
It was expected that performance would take a hit since deploying smart contracts is an intensive process, and the test in question involves deploying a new smart contract once every four operations.



(a) Throughput Variation

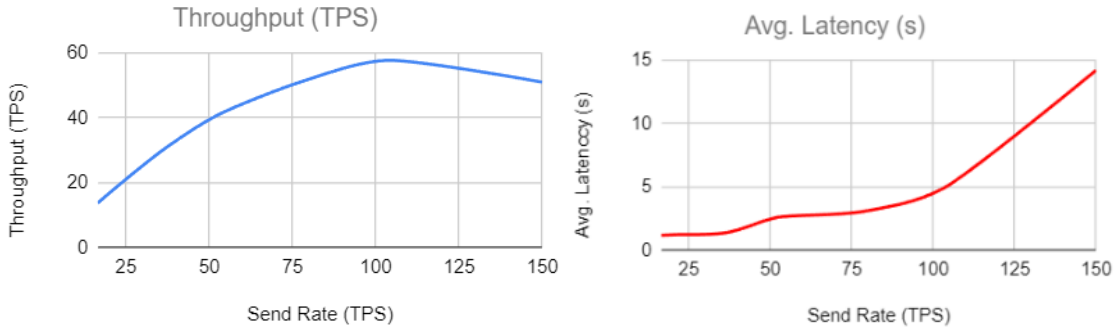
(b) Latency Variation

Figure 6.2: Rental Benchmark

6.2.3 Sale Use Case

The benchmark for the sale use case resulted in slightly lower values to those of the rental use case. The maximum throughput is reached at a transaction send rate of around 100 TX/s, with values nearing 58 TX/s. For higher send rate values, the throughput decreased

considerably, stagnating at around 51 TX/s. As for latency, slightly better results were observed.
This results were expected since the sale use case involves deploying a new smart contract once every three operations, and has the additional ownership transfer overhead when compared to the rental use case.



(a) Throughput Variation

(b) Latency Variation

Figure 6.3: Sale Benchmark

6.3 Software Development Life Cycle

If a system of this caliber were to be implemented, we would need to go far beyond the work done in this chapter to validate the correction and performance of the developed solution. The best and safest software development practices must be applied. Test-Driven Development (TDD) should be one of the best practices for handling the application requirements. TDD will allow the creation of robust code specifications before any applicational code is developed. This can be done, for instance, through the definition of early metamodels. Contract drafts can start in parallel with the technical staff and be written in natural language alongside code that validates the contract's requirements. Furthermore, such an ambitious system will require development tactics that leverage system decoupling. Beyond unit testing, the architecture design should be mindful of designing software artifacts that can be run without interface coupling. Another great bonus is formal verification of smart contracts. The major issue with unit testing is that it can't prove correctness for input values that are not part of the sample. Formal verification, however, can formally prove that a smart contract satisfies requirements for an infinite range of executions without running the contract at all [23]. Formal verification is of great importance when assessing correctness of safety-critical systems whose failure can have devastating consequences, such as death, injury, or financial ruin. Then, integration testing follows. Differently from what was done at section 6.1, we should have an actual dedicated integrated environment where developers can deploy subsystems and components to drive the application's end-to-end validation, with the goal of debugging the

system with all parts communicating between them.
At the quality assurance phase, we would expect a dedicated testing team to pick up the application and apply a robust set of tests to validate the User Acceptance Criteria on a system close to the final one. We should validate expected user behaviors and unexpected or unintended interactions with data samples that replicate the ones from a real-world application. After quality assurance, a production validation phase should mostly care about the non-functional requirements, putting the application under stress and performance tests, validating security expectations, and guaranteeing that the system is business-ready. Finally, we should be able to deploy the system into the final environment. One of the most important non-functional requirements for this phase is observability. The system should generate enough information or allow for a comprehensive probing that supports any type of analysis from the engineering teams.

6.4 Relevant Considerations

6.4.1 Smart Contract Upgrading and Bug Resolution

One of the most critical requirements of a complete evaluation is to ensure the correctness and integrity of the solution, since we are dealing with a blockchain solution where the deployed code is, by default, immutable and tamper-proof. So guarantee the fundamental characteristics of smart contracts. It is reasonable to assume, however, that even the most complete evaluation may have blind spots. There is always the possibility that even after extensive testing, the solution deployed in production may contain bugs and exploits. If this occurs, it is critical that once the bug is discovered, the code is immediately fixed or replaced, and that any type of data that may have been corrupted by it is addressed or reverted. But how can we do this if the smart contracts that are deployed are immutable? Actually, there are several measures and workarounds we can resort to. Some are more generic, others use specific features and functionalities of Ethereum or are only feasible on permissioned networks. There is no single right answer, however. Each philosophy has its pros and cons, and it really depends on the situation. Let's begin by looking at some of the main alternatives available on the public Ethereum network [23].

6.4.1.1 Manual Migration

Migration is the process of deploying a new smart contract, independent of the previous contract, and manually migrating the data to the new one. This process may not be simple depending on the complexity of the contract, and in public networks it also has the major disadvantage of being very expensive in terms of gas fees. Moreover, it may be necessary to add endpoints to the interface that allow or facilitate the migration of data to a new version. A good side of it is that in the data migration process it may be possible to filter

the data, allowing the discarding or fixing of invalid or corrupted data. In the context of permissioned networks, this may be an alternative to consider, as there are no transaction fees and there is always the possibility of interrupting the system for maintenance, which can greatly facilitate the process.

6.4.1.2 The Registry Pattern

The Registry Pattern is, in fact, one of the alternatives made available in the developed solution. It also complements the migration approach really well. It involves having a registry contract whose function is to assign the most recent smart contract address to a service or a group of services, acting like a name resolver. The registry smart contract provides functions to change the address of a service, allowing us to deploy new smart contracts with upgraded code. Thus, when it is necessary to make a call to a certain service, this approach involves first checking in the registry which smart contract address we should use. This solution can be quite viable in cases where the contracts to be upgraded do not contain an internal state. Since it involves deploying an entirely new smart contract, the state of the previous one will not persist unless this approach is complemented with some type of migration. In the developed solution, this is the pattern implemented by the *ContractNameService* smart contract. Although it can be used in this context, it was implemented mostly with the purpose of facilitating communication between the system's smart contracts.

6.4.1.3 Data Separation

Another method for upgrading smart contracts is to separate business logic and data storage into separate contracts. This means users interact with the logic contract, which contains the code, while data is stored in the storage contract [23]. The logic contract holds the storage contract's address and interacts with it to write and read data. The storage contract is also configured with the latter's address at deployment, so it guarantees that only the logic contract may update the data. The storage contract should be mostly immutable, only containing basic functions to replace the logic contract address it points to. By deploying a completely new logic smart contract and making the storage counterpart point to it, we are effectively updating the code while keeping the storage intact. Thus, this method already comes very close to the concept of code upgrading. It requires some management since multiple contracts are at stake, which demands rigorous access control. However, it should be relatively simple to implement.

6.4.1.4 The Proxy Pattern

The final method available on public networks, and arguably the most popular one, are proxy deployments. The Proxy pattern is the strategy that is most loyal to the concept of

smart contract upgrading. It also uses data separation to keep business logic and data in separate contracts. In this case, a proxy contract shall be deployed next to our implementation contract (the one which contains the code logic), which will serve as the storage contract. In order to understand how the proxy contract works, we need to take a look at Solidity's *delegatecall* function. *Delegatecall* is a low level operation that allows a contract to call another contract's function, while the actual code execution happens in the context of the calling contract. The proxy contract stores the address of the logic contract and delegates all function calls to the implementation contract, using the *delegatecall* function. This means that the proxy contract will use the code specified in the implementation contract, but will store all the state locally. Simply putting, the users will send operations to the proxy contract, which then forwards them to the implementation contract. Upon execution, the implementation contract returns the result and data, which is stored within the proxy. Like in the data separation method, upgrading a smart contract with a proxy is as straightforward as deploying a new smart contract containing the updated code, and making the proxy point to it.
The proxy contract itself should be simple and generic, as it should not contain any application logic - it just requires writing a custom *fallback* function specifying how the proxy contract should handle function calls it does not support. Proxy patterns eliminate the burden of migrating data to new contracts, and are not invasive to developers as there's no need to make any changes to the smart contracts implementing the business logic, apart from some conflict regarding function selector clashes.

6.4.1.5 Considerations for Upgrading Smart Contracts

Although being able to upgrade smart contracts brings flexibility and the ability to fix unexpected vulnerabilities, it also have several implications regarding the system's trustability. The core characteristic of smart contracts that makes them highly relevant, not only in the context of this project but for any blockchain solution, is precisely their immutability. Smart contracts were designed to be unbreakable and non-violable. They offer a solution for situations where entities need to conduct business but don't necessarily trust each other. A common threat seen in all these approaches involves the need for an entity or a set of entities with privileged rules capable of altering the contract terms. There is a risk that users engaged with the old version of the contract may not agree with the terms of the new version. This is particularly relevant for the proxy pattern, as it forces users to adopt new versions whether they agree or not. Therefore, if the solution were to be implemented in a production environment, this issue would need to be thoroughly studied and analyzed, as it is not possible to have the best of both worlds.

6.4.2 Permissioned Networks Upgrading Mechanisms

In permissioned blockchain environments, smart contract upgrading and bug resolution is more flexible than in public blockchains. Since these are controlled by a consortium or organization with established governance protocols, they can allow for system modifications without compromising trust. For instance, frameworks like Hyperledger Fabric enable the redeployment or upgrading of smart contracts through governance consensus or administrative intervention. Moreover, the governance/permissioning models in permissioned blockchains can incorporate access control measures, such as enabling privileged roles to deploy new contract versions or perform maintenance operations. These mechanisms provide the flexibility needed for evolving systems while ensuring accountability. Lastly, permissioned networks often support features like state management and checkpointing, where the system can be rolled back to a previous stable state if a critical error happens or corrupted data is detected.

7

Conclusion and Final Remarks

7.1 Governance and Executive Implications and Challenges

Bringing blockchain technology to the real estate market at a nation-wide level presents several governance and executive implications and challenges. The purpose of this final section is to raise awareness of various issues that must be addressed if a blockchain solution is ever planned to be integrated into the real estate sector, while encouraging further study and exploration. By identifying and understanding these issues, stakeholders can develop more effective strategies and solutions.

7.1.1 Legal Limitations

The first barrier to adopting blockchain in real estate is the significant disconnect between existing laws and emerging technologies. The current legal framework is not designed to accommodate the intricacies of blockchain, resulting in incomplete translations of legal aspects into the blockchain or the need for workaround solutions. As seen in section 2.8, a prime example of this is the GDPR. The GDPR is a set of regulations aimed at safeguarding citizen privacy with a strong humanitarian focus. But its principles clash directly with the immutable nature of blockchain, creating a paradox that is difficult to resolve under the current legal paradigm.
As technology advances, an essential question arises: will future laws be adequately equipped to handle the nuances of blockchain technology? If the answer is no, what alternatives exist? Emerging technologies like Digital Asset's Daml and Canton are attempting to bridge this gap by fitting within the current legal framework. However, these solutions may not fully address the complexities involved. Comprehensive legal reforms may be necessary to create an environment where blockchain can thrive without legal impediments.

7.1.2 Legal Automation through Smart Contracts

Even assuming that existing legal frameworks can adapt to accommodate blockchain, this transition would imply a significant shift in how laws are enforced with this technology.

Traditionally, laws require enforcement by human agents, such as judges, police officers, and regulatory officials. However, with blockchain, smart contract code becomes definition of the law itself, executed automatically without human intervention. Smart contracts bring a way to execute these laws autonomously, ensuring that transactions and agreements are carried out precisely as coded. This shift towards automation poses significant questions about accountability and oversight. For instance, if a smart contract operates incorrectly or is exploited, determining responsibility becomes complex. Is the developer who wrote the contract at fault, or does the responsibility lie with the users who interacted with it? Or should the network's governance model be accountable? Furthermore, in traditional legal systems, human agents can interpret the law with a degree of flexibility, considering unique circumstances in each case. Smart contracts, on the other hand, execute predetermined rules without exception. This rigidity may lead to unintended consequences if the contract conditions do not cover every possible scenario.
Another issue is the need for a mechanism to resolve disputes arising from smart contract execution. In a conventional system, disputes are resolved through judicial processes, but with smart contracts, the outcome is strictly determined by the code. Therefore, integrating an arbitration or dispute resolution process within the blockchain ecosystem is crucial to address conflicts that may arise from automated legal enforcement.

7.1.3 Identity Management

A critical component of such a system is a robust identity management framework. In this work, we propose a simple and straightforward identity system, fully managed on-chain through permission smart contracts and assisted by a public ethereum wallet client, Metamask. But in a real world setting, a much richer and deeper architecture for Public Key Infrastructure (PKI) is desirable. The responsibility for maintaining the PKI must be clearly defined. How PKI is implemented can significantly impact the system's security and usability. It may be beneficial to base account cryptographic keys on a citizen's unique identifier, like the eIDAS or other national identification systems. This effort contributes to generalization of authentication procedures and ensures that each citizen has a unique key, preventing duplication.
Then, other concerns arise: what happens if a key is lost? Should we implement multi-signature keys that allow asset transfers even if a key is lost? Alternatively, should keys be stored in a secure wallet protected by a username and password? Each of these solutions bring implications for user convenience and system security.

7.1.4 Physical Infrastructure Topology and Maintenance

The maintenance of the physical infrastructure for an enterprise blockchain system is another significant consideration. Beyond determining which organizations will host nodes and what information each node will access, it is necessary to decide who will

manage the infrastructure of these nodes and where they will be physically located. It is reasonable to assume that there are organizations that wish to participate in the network but do not necessarily have the personnel with the necessary qualifications to manage a blockchain node. A conservative approach might involve having all nodes actually managed and controlled by the government, potentially hosted in public service offices.
There are more considerations regarding the infrastructure of the network that should be made. For example, regarding the network topology – we might want to include other services, systems, and technologies adjacent to the blockchain, such as an InterPlanetary File System (IPFS) network, sidechains, firewalls, abstraction layers, among others.
Additionally, it is important to consider the storage requirements of the blockchain. Is it necessary for all nodes to store the entire blockchain? Depending on the blockchain technology, it is possible to implement checkpoints or pruning mechanisms to reduce the storage burden on individual nodes.
Moreover, we need to address the issue of scalability and latency. How will the network handle a growing number of transactions and users? Should we employ sharding or layer-2 solutions to improve throughput? These decisions will significantly affect the performance and resilience of the blockchain system. Ensuring that the physical infrastructure is capable of supporting these technical requirements is essential for the long-term success of the blockchain deployment.

7.1.5 Interoperability Support

Supporting interoperability between different blockchains and legacy systems can potentially be a viable approach for a more seamless and gradual transition. That is, instead of completely overhauling existing systems, blockchain can be used as a way of unifying data that is scattered across different platforms, creating a shared environment with the gradual onboarding of multiple stakeholders. In practical terms, this means using blockchain bridging mechanisms to connect disparate systems. This approach not only enables the synchronization of data but also ensures that each participant can maintain their existing infrastructure while benefiting from blockchain's characteristics. This is an alternative that is yet to see more exploration in real-world applications.

7.2 Conclusion

This dissertation had the primary objective of understanding and demonstrating the potential impact that blockchain technology could have on the real estate sector in the future. Along the way, we explored multiple international governmental and business initiatives, examined how country regulations may affect the realization of this vision, analyzed how stakeholders and processes would be impacted by such autonomous systems, and considered how tokenization can be utilized to revitalize the real estate market. We also investigated how fractional ownership of assets could be feasible in a blockchain environment and discussed relevant considerations that should be taken in a production scenario.

Blockopoly introduced a smart contract architecture that enforces permissioning mechanisms for the involved stakeholders and employs a multi-signature approach to real-world contracts for processes related to property sales, rentals, and loans. The benefits of this approach include its generic nature, modularity, and high capacity for functionality expansion. On the other hand, its weaknesses include resource-intensive deployment of smart contracts, making it possibly unsuitable for some public networks, such as Ethereum's mainnet, due to the high transaction costs. Another limitation of the study was its reliance on a single blockchain framework (Ethereum / Hyperledger Besu), which, although highly versatile, may not generalize across all blockchain platforms.

Finally, this research contributes to a better understanding of how smart contracts can be leveraged in the real estate industry. Although far from full realization, it serves as an incentive for the disruptive movement of blockchain in the world.

Bibliography

[1] K. M. Alam et al. "A Blockchain-based Land Title Management System for Bangladesh". In: *Journal of King Saud University - Computer and Information Sciences* (2022-06). ISSN: 22131248. DOI: 10.1016/j.jksuci.2020.10.011 (cit. on p. 24).

[2] T. Ali et al. "A transparent and trusted property registration system on permissioned blockchain". In: *International Conference on Advances in the Emerging Computing Technologies, AECT 2019*. Institute of Electrical and Electronics Engineers Inc., 2020-02. ISBN: 9781728144528. DOI: 10.1109/AECT47998.2020.9194222 (cit. on p. 23).

[3] R. Arede. *Blockopoly Repository*. URL: https://github.com/rmarede/Blockopoly (cit. on p. 76).

[4] ATLANT Platform. "ATLANT Real Estate Platform". URL: https://medium.com/@atlantio/atlant-real-estate-platform-c2b4ef85079d (cit. on p. 29).

[5] R. Bennett et al. "Hybrid approaches for smart contracts in land administration: Lessons from three blockchain proofs-of-concept". In: *Land* (2021-02). ISSN: 2073445X. DOI: 10.3390/land10020220 (cit. on pp. 25, 26).

[6] R. M. Bennett, M. Pickering, and J. Sargent. "Transformations, transitions, or tall tales? A global review of the uptake and impact of NoSQL, blockchain, and big data analytics on the land administration sector". In: *Land Use Policy* (2019-04). ISSN: 02648377. DOI: 10.1016/j.landusepol.2019.02.016 (cit. on p. 31).

[7] Binaryx. *Binaryx Website*. URL: https://www.binaryx.com/ (cit. on p. 29).

[8] R. G. Brown. "The Corda Platform: An Introduction". 2018 (cit. on p. 16).

[9] V. Buterin. *Ethereum: A Next-Generation Smart Contract and Decentralized Application Platform*. Tech. rep. (cit. on p. 11).

[10] A. Castellanos. *Digitalization of Land Records: From Paper to Blockchain*. Tech. rep. 2018 (cit. on p. 27).

[11] J. Chen and S. Micali. "Algorand". 2016-07. URL: http://arxiv.org/abs/1607.01341 (cit. on p. 16).

[12] F. Coelho and G. Younes. *The GDPR-Blockchain paradox: A work around*. Tech. rep. 2018. URL: https://www.researchgate.net/publication/329656420 (cit. on p. 15).

[13] ConsenSys. *Tessera Documentation*. URL: https://docs.tessera.consensys.io (cit. on p. 39).

[14] Consensys. "Quorum Whitepaper". 2018. URL: https://github.com/ethereum/wiki/wiki/White-Paper (cit. on pp. 11, 16).

[15] C. Crittenden. *Blockchain in California: A Roadmap*. Tech. rep. California Blockchain Working Group, 2020 (cit. on p. 8).

[16] Deloitte. *Are token assets the securities of tomorrow?* Tech. rep. 2019 (cit. on p. 14).

[17] Deloitte. *Blockchain in commercial real estate*. Tech. rep. 2017 (cit. on pp. 2, 5).

[18] O. Dib et al. "Consortium Blockchains: Overview, Applications and Challenges". In: *International Journal on Advances in Telecommunications* 2 (2018) (cit. on pp. 9, 32).

[19] Digital Asset. *Daml SDK Documentation v2.8.7*. Tech. rep. 2023 (cit. on p. 17).

[20] e-estonia. *Estonian blockchain technology FAQ*. Tech. rep. (cit. on p. 26).

[21] e-estonia. *KSI blockchain in Estonia*. Tech. rep. (cit. on p. 26).

[22] eForms. *Residential Lease Agreement Template*. Tech. rep. 2024. URL: https://eforms.com/rental/ (cit. on p. 65).

[23] Ethereum Foundation. *Ethereum Docs*. URL: https://ethereum.org/developers/docs (cit. on pp. 36, 80–82).

[24] European Commission. *Regulation (EU) 2023/1114 of the European Parliament and of the Council of 31 May 2023 on markets in crypto-assets (MiCA)*. 2023. URL: http://data.europa.eu/eli/reg/2023/1114/oj (cit. on p. 14).

[25] C. Fan et al. "Performance Analysis of Hyperledger Besu in Private Blockchain". In: *Proceedings - 4th IEEE International Conference on Decentralized Applications and Infrastructures, DAPPS 2022*. Institute of Electrical and Electronics Engineers Inc., 2022, pp. 64–73. ISBN: 9781665491723. DOI: 10.1109/DAPPS55202.2022.00016 (cit. on pp. 77, 78).

[26] P. Freni, E. Ferro, and R. Moncada. "Tokenomics and blockchain tokens: A design-oriented morphological framework". In: *Blockchain: Research and Applications* (2022-03). ISSN: 26669536. DOI: 10.1016/j.bcra.2022.100069 (cit. on p. 13).

[27] R. M. Garcia-Teruel. "Legal challenges and opportunities of blockchain technology in the real estate sector". In: *Journal of Property, Planning and Environmental Law* (2020-07). ISSN: 25149407. DOI: 10.1108/JPPEL-07-2019-0039 (cit. on pp. 1, 4, 22, 26, 33).

[28] M. Graglia and C. Mellon. *Blockchain and Property in 2018*. Tech. rep. 2018. URL: http://direct.mit.edu/itgg/article-pdf/12/1-2/90/705267/inov_a_00270.pdf (cit. on p. 32).

[29] A. Gupta et al. "Tokenization of real estate using blockchain technology". In: *Applied Cryptography and Network Security Workshops*. Vol. 12418 LNCS. Springer Science and Business Media Deutschland GmbH, 2020, pp. -9. ISBN: 9783030616373. DOI: 10.1007/978-3-030-61638-0{_}5 (cit. on pp. 1, 21, 23, 29).

[30] H. Hazeem and E. AlBurshaid. "Fragmented Data Landscape and Data Asymmetries in the Real Estate Industry". In: *Blockchain in Real Estate*. Singapore: Springer Nature Singapore, 2024, pp. 179–205. DOI: 10.1007/978-981-99-8533-3{_}10 (cit. on pp. 1, 20).

[31] Hyperledger Foundation. *An Introduction to Hyperledger*. Tech. rep. 2018 (cit. on p. 16).

[32] Hyperledger Foundation. *Besu Documentation*. URL: https://besu.hyperledger.org (cit. on pp. 11, 39).

[33] A. J. Jiménez Clar. *The Real Estate Information: A Way to Standardization*. Tech. rep. 2001 (cit. on pp. 1, 15, 20).

[34] S. Joshi and A. Choudhury. "Tokenization of Real estate Assets Using Blockchain". In: *International Journal of Intelligent Information Technologies* (2022). ISSN: 15483665. DOI: 10.4018/IJIIT.309588 (cit. on p. 21).

[35] A. Jusić. "Privacy between Regulation and Technology: GDPR and the Blockchain". In: *IUS Law Journal* Vol 1.No 1 (2022), pp. 47–59. ISSN: 28310047. DOI: 10.21533/iuslawjournal.v1i1.9 (cit. on p. 15).

[36] I. Karamitsos, M. Papadaki, and N. B. A. Barghuthi. "Design of the Blockchain Smart Contract: A Use Case for Real Estate". In: *Journal of Information Security* (2018). ISSN: 2153-1234. DOI: 10.4236/jis.2018.93013 (cit. on p. 22).

[37] A. Krause and C. A. Lipscomb. "The data preparation process in real estate: Guidance and review". In: *Journal of Real Estate Practice and Education* 19.1 (2016), pp. 15–42. ISSN: 15214842. DOI: 10.1080/10835547.2016.12091756 (cit. on pp. 1, 20).

[38] V. N. Kustov and E. S. Selanteva. "Mutual Recognition Mechanism Based on DVCS Oracle in the Blockchain Platform". In: 2022, pp. 81–103. DOI: 10.4018/978-1-7998-8697-6.ch005 (cit. on p. 13).

[39] Lantmäteriet et al. *The Land Registry in the Blockchain - Testbed*. Tech. rep. 2017 (cit. on p. 25).

[40] LaProp. *Tokenized Real Estate*. Tech. rep. 2022 (cit. on p. 29).

[41] J. M. Lourenço. *The NOVAthesis LaTeX Template User's Manual*. NOVA University Lisbon. 2021. URL: https://github.com/joaomlourenco/novathesis/raw/main/template.pdf (cit. on p. i).

[42] T. Moura and I. Azevedo. "Design Patterns for Ethereum Smart Contracts". PhD thesis. 2020 (cit. on p. 43).

[43] S. Nakamoto. *Bitcoin: A Peer-to-Peer Electronic Cash System*. Tech. rep. 2008. URL: www.bitcoin.org (cit. on p. 7).

[44] C. Nevile et al. *Enterprise Ethereum Alliance Client Specification v6*. Tech. rep. 2020 (cit. on p. 37).

[45] Object Management Group. *Object Constraint Language v2.4*. 2014. URL: http://www.omg.org/spec/OCL/2.4 (cit. on p. 41).

[46] OpenZeppelin. *Gnosis Multisig*. URL: https://github.com/OpenZeppelin/gnosis-multisig (cit. on p. 45).

[47] Propy. *Propy Website*. URL: https://propy.com/ (cit. on p. 27).

[48] Propy. "Propy Whitepaper 2.0". 2021 (cit. on pp. 27, 28).

[49] R3. *Blockchain 101*. URL: https://r3.com/blockchain-101/ (cit. on p. 35).

[50] RealT. "Legally Compliant Ownership of Tokenized Real Estate" (cit. on pp. 23, 29).

[51] Rentberry. "Rentberry Whitepaper". 2018 (cit. on pp. 22, 29).

[52] REX Protocol. *REX Protocol Public Docs*. URL: https://rex-protocol.gitbook.io/rex-protocol-public-docs/ (cit. on p. 29).

[53] A. Saari, J. Vimpari, and S. Junnila. "Blockchain in real estate: Recent developments and empirical applications". In: *Land Use Policy* (2022-10). ISSN: 02648377. DOI: 10.1016/j.landusepol.2022.106334 (cit. on p. 23).

[54] S. Sen, S. Mukhopadhyay, and S. Karforma. "A Blockchain based Framework for Property Registration System in E-Governance". In: *International Journal of Information Engineering and Electronic Business* (2021-08). ISSN: 20749023. DOI: 10.5815/ijieeb.2021.04.03 (cit. on p. 24).

[55] L. Shin. "Republic Of Georgia To Pilot Land Titling On Blockchain, BitFury". In: *Forbes* (2016). URL: https://www.forbes.com/sites/laurashin/2016/04/21/republic-of-georgia-to-pilot-land-titling-on-blockchain-with-economist-hernando-de-soto-bitfury/?sh=27a76a2f44da (cit. on p. 26).

[56] L. Shin. "The First Government To Secure Land Titles On The Bitcoin Blockchain Expands Project". In: *Forbes* (2017). URL: https://www.forbes.com/sites/laurashin/2017/02/07/the-first-government-to-secure-land-titles-on-the-bitcoin-blockchain-expands-project/?sh=7c4e1894dcdc (cit. on p. 26).

[57] M. Shuaib et al. "Blockchain-based framework for secure and reliable land registry system". In: *Telkomnika (Telecommunication Computing Electronics and Control)* (2020-10). ISSN: 23029293. DOI: 10.12928/TELKOMNIKA.v18i5.15787 (cit. on p. 4).

[58] M. Shuaib et al. "Current Status, Requirements, and Challenges of Blockchain Application in Land Registry". In: *International Journal of Information Retrieval Research* (2022-08). ISSN: 2155-6377. DOI: 10.4018/ijirr.299934 (cit. on pp. 1, 4, 5, 26).

[59] M. Shuaib et al. *Identity Model for Blockchain-Based Land Registry System: A Comparison*. 2022. DOI: 10.1155/2022/5670714 (cit. on p. 1).

[60] A. Sriram. *California DMV puts 42 million car titles on blockchain to fight fraud*. 2024-07. URL: https://www.reuters.com/technology/california-dmv-puts-42-million-car-titles-blockchain-fight-fraud-2024-07-30/ (cit. on p. 8).

[61] W. Stallings. *Network Security Essentials: Applications and Standards*. Prentice Hall, 2011, p. 417. ISBN: 0136108059 (cit. on p. 10).

[62] Statista. *Real Estate Worldwide*. Tech. rep. URL: https://www.statista.com/outlook/fmo/real-estate/worldwide (cit. on p. 1).

[63] N. Szabo. *Smart Contracts*. 1994. URL: https://www.fon.hum.uva.nl/rob/Courses/InformationInSpeech/CDROM/Literature/LOTwinterschool2006/szabo.best.vwh.net/smart.contracts.html (cit. on p. 10).

[64] Tokeny. *T-REX Protocol (Token for Regulated EXchanges) Whitepaper*. Tech. rep. 2023. URL: https://github.com/TokenySolutions (cit. on p. 14).

[65] USAID. *Honduras | LandLinks*. Tech. rep. 2011. URL: https://www.land-links.org/country-profile/honduras/ (cit. on p. 27).

[66] P. Voigt and A. von dem Bussche. *The EU General Data Protection Regulation (GDPR)*. Cham: Springer International Publishing, 2017. ISBN: 978-3-319-57958-0. DOI: 10.1007/978-3-319-57959-7 (cit. on p. 15).

[67] World Bank. *Doing Business Report - Registering Property*. Tech. rep. 2019. URL: https://archive.doingbusiness.org/en/data/exploretopics/registering-property (cit. on p. 26).

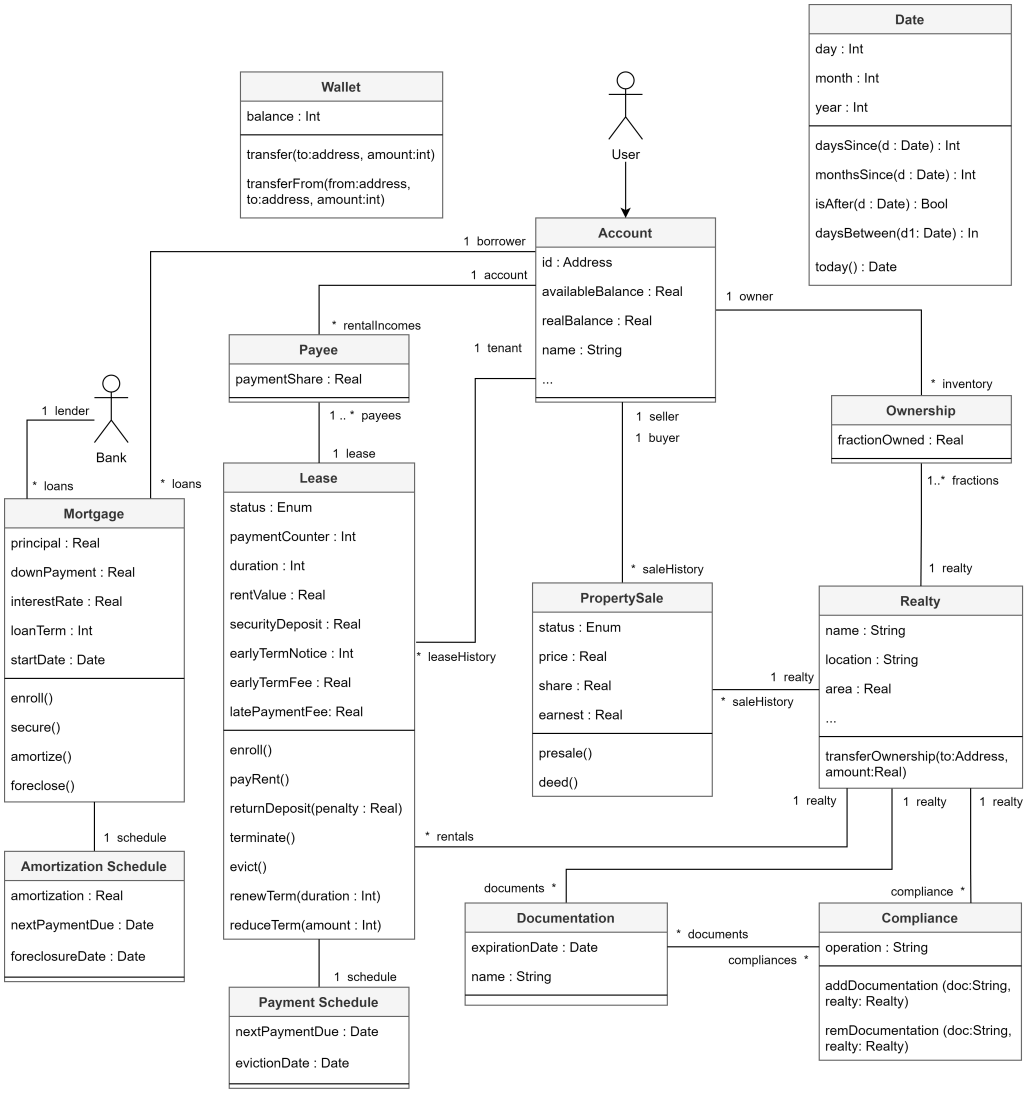
[68] D. Yaga et al. "Blockchain Technology Overview". In: (2019-06). DOI: 10.6028/NIST.IR.8202 (cit. on pp. 9, 10).

[69] Z. Zheng et al. "An Overview of Blockchain Technology: Architecture, Consensus, and Future Trends". In: *Proceedings - 2017 IEEE 6th International Congress on Big Data, BigData Congress 2017*. Institute of Electrical and Electronics Engineers Inc., 2017-09, pp. 557–564. ISBN: 9781538619964. DOI: 10.1109/BigDataCongress.2017.85 (cit. on pp. 7, 8).

A

System Metamodel

The high-level metamodel of the system architecture mentioned in section 4.6.



```
{context Realty
  inv: fractions.fractionOwned->sum() = 1.0
  inv: fractions->isUnique(f | f.owner)
}
```

Figure A.1: Realty OCL

- The sum of a realty's fractions owned by different entities must be equal to 1 (100%);
- Each owner must be unique.

```
{context Realty::transferOwnership(fraction:Real, from:Address, to:Address)
  pre: fractions->select(f | f.owner.address=from).fractionOwned > fraction
  post: fractions->select(f | f.owner.address=from).fractionOwned =
      fractions->select(f | f.owner.address=from).fractionOwned@pre - fraction
      and fractions->select(f | f.owner.address=to).fractionOwned =
      fractions->select(f | f.owner.address=to).fractionOwned@pre + fraction
}
```

Figure A.2: Transfer Ownership OCL

- The account giving up the ownership must first have a fractional ownership greater or equal to the share being transferred;
- After the transaction, the ownership share must have been transferred from the sender to the receiver.

```
{context PropertySale
  inv: status=PENDING or status=AGREED or status=COMPLETE or status=WITHDRAWN
  inv: if status=AGREED then realty.fractions->select(f | f.owner =
      issuer).fractionOwned >= fraction endif
  inv: issuer <> buyer
}
```

Figure A.3: Property Sale OCL

- A sale can have the status PENDING, AGREED, COMPLETED or WITHDRAWN;
- If it has the status AGREED, then the seller must still have ownership equal to or greater than the share being sold;
- The issuer of a sale must always be different from the buyer.

```
1  {context PropertySale::presale()
2    pre: status=PENDING and buyer.availableBalance >= earnest and
         seller.inventory->one(o|o.realty=realty).fractionOwned > share
3    post: status=AGREED and buyer.availableBalance = buyer.availableBalance@pre -
         earnest
4  }
```

Figure A.4: Presale OCL

- For a sale to be pre-agreed, it must be PENDING, the buyer's balance must be equal or greater to the earnest amount, and the seller's ownership equal to or greater than the share being sold;
- After a sale has been pre-agreed on, its status must have changed to AGREED and the earnest amount must be locked from the buyer's balance.

```
1  {context PropertySale::deed()
2    pre: status=AGREED and buyer.availableBalance >= price and
         realty.fractions->select(f | f.owner=seller).fractionOwned > share
3
4    post: status=COMPLETE and
5      let sellerOwnershipBefore: Real = realty.fractions->select(f |
           f.owner=seller).fractionOwned@pre in
6      let sellerOwnershipAfter: Real = realty.fractions->select(f |
           f.owner=seller).fractionOwned in
7      let buyerOwnershipBefore: Real = realty.fractions->select(f |
           f.owner=buyer).fractionOwned@pre in
8      let buyerOwnershipAfter: Real = Real = realty.fractions->select(f |
           f.owner=buyer).fractionOwned in
9
10     sellerOwnershipAfter = sellerOwnershipBefore - share and
11     buyerOwnershipAfter = buyerOwnershipBefore + share and
12     buyer.realBalance = buyer.realBalance@pre - price and
13     seller.realBalance = seller.realBalance@pre + price * (1 - comission) and
14     seller.availableBalance = seller.availableBalance@pre + price *
           (1-comission) and
15     realtor.realBalance = realtor.realBalance@pre + price * comission and
16     realtor.availableBalance = realtor.availableBalance@pre + bid.amount *
           comission
17 }
```

Figure A.5: Deed OCL

- In order to proceed with the deed, the sale must be AGREED, the buyer's balance must be equal or greater to the sale price, and the seller's ownership equal to or greater than the share being sold;
- Once the deed is successful, the sale status becomes COMPLETE. The ownership share is transferred from the seller to the buyer, and the sale price is transferred

from the buyer to the seller (minus the commission). If applicable, a commission is transferred to the realtor associated with the sale.

```
{context PropertySale::withdraw(penalty:Real)
  pre: status=AGREED and penalty <= earnest

  post: status=WITHDRAW and buyer.availableBalance = buyer.availableBalance@pre
      + (earnest-penalty) and seller.availableBalance =
      seller.availableBalance@pre + penalty and seller.realBalance =
      seller.realBalance@pre + penalty
}
```

Figure A.6: Withdraw OCL

- Only a sale which was already AGREED on may be withdrawn;
- When withdrawing a sale, a penalty may be specified, which will be decreased the buyer's locked balance (equivalent to the earnest amount) and transferred to the seller. The remaining amount will be returned to the buyer.

```
{context Lease
  inv: status=PENDING or status=ACTIVE or status=COMPLETE or status=TERMINATED
}
```

Figure A.7: Lease OCL

- A lease can have the status PENDING, ACTIVE, COMPLETE or TERMINATED;

```
{context Lease::enroll()
  pre: status=PENDING and tenant.availableBalance >= securityDeposit + rentValue

  post: status=ACTIVE and tenant.availableBalance -= securityDeposit + rentValue
      and tenant.realBalance -= rentValue and paymentCounter=1 and
    payees->forall(l | l.account.availableBalance =
        l.account.availableBalance@pre + price *
        l.account.inventory->one(o|o.realty=realty).fractionOwned) and
    payees->forall(l | l.account.realBalance = l.account.realBalance@pre + price
        * l.account.inventory->one(o|o.realty=realty).fractionOwned)
}
```

Figure A.8: Enroll

- Only a PENDING lease may be enrolled on;
- To enroll, the tenant must have a balance equal or greater than the sum of the security deposit and the rent value;

- After enrolling, the lease becomes ACTIVE, the payment counter starts and the first month of rent is divided between the payees according to the share defined in the contract terms.

```
1 {context Lease::payRent()
2   pre: status=ACTIVE and tenant.availableBalance >= rentValue
3
4   post: let totalPayment : Int = latePaymentFee *
        daysSince(schedule.nextPaymentDue) + rentValue in
5   tenant.availableBalance -= totalPayment and tenant.realBalance -= totalPayment
        and paymentCounter=paymentCounter += 1 and schedule.nextPaymentDate += 1
        month and schedule.evictionDate += 1 month and
6    payees->forall(l | l.account.availableBalance =
        l.account.availableBalance@pre + totalPayment *
        l.account.inventory->one(o|o.realty=realty).fractionOwned) and
7    payees->forall(l | l.account.realBalance = l.account.realBalance@pre +
        totalPayment * l.account.inventory->one(o|o.realty=realty).fractionOwned)
8    and if paymentCounter=duration status=COMPLETE else status=ACTIVE
9 }
```

Figure A.9: Pay Rent

- The tenant may pay rent on an ACTIVE lease. His balance must be equal or greater than the rent value;
- The rent amount is divided between the payees according to the share defined in the contract terms;
- The payment counter is increased, and if reaches the term duration, the lease becomes COMPLETE.

```
1 {context Lease::returnDeposit(penalty:Real)
2   pre: status = COMPLETE and penalty <= securityDeposit
3
4   post: status = TERMINATED and
5    tenant.realBalance = tenant.realBalance@pre - penalty and
6    tenant.availableBalance = tenant.availableBalance@pre + (securityDeposit -
        penalty) and
7    landlords->forall(l | l.account.availableBalance =
        l.account.availableBalance@pre + penalty *
        l.account.inventory->one(o|o.realty=realty).fractionOwned) and
8    landlords->forall(l | l.account.realBalance = l.account.realBalance@pre +
        penalty * l.inventory->one(o|o.realty=realty).fractionOwned)
9 }
```

Figure A.10: Return Deposit

- The security deposit must be returned once the lease is COMPLETE;
- After the security deposit is returned, the lease becomes TERMINATED;

- A penalty equal to or less than the security deposit may be specified. If applicable, the penalty is divided between the payees according to the share defined in the contract terms. The remaining amount is returned to the tenant.

```
1 {context Lease::terminate()
2   pre: status=ACTIVE
3   post: status=COMPLETE
4 }
```

Figure A.11: Terminate

- Only an ACTIVE lease may terminated forcibly;
- Terminating the lease leaves it with the COMPLETE status.

```
1 {context Lease::evict()
2   pre: status=ACTIVE and Date.today().isAfter(schedule.evictionDate)
3   post: status=TERMINATED and duration=duration@pre-months and
4     tenant.realBalance = tenant.realBalance@pre - securityDeposit and
5     landlords->forall(l | l.account.availableBalance =
          l.account.availableBalance@pre + securityDeposit *
          l.account.inventory->one(o|o.realty=realty).fractionOwned) and
6     landlords->forall(l | l.account.realBalance = l.account.realBalance@pre +
          securityDeposit * l.inventory->one(o|o.realty=realty).fractionOwned)
7 }
```

Figure A.12: Evict

- A tenant of an ACTIVE lease may be evicted if he fails to meet the payment schedule.
- Upon eviction, the lease becomes TERMINATED and the security deposit is divided between the payees according to the share defined in the contract terms.

```
1 {context Lease::reduceTerm(amount:Int)
2   pre: status=ACTIVE
3   post: duration = duration@pre-months - amount and
4     tenant.realBalance = tenant.realBalance@pre - earlyTermFee and
5     tenant.availableBalance = tenant.availableBalance@pre - earlyTermFee and
6     landlords->forall(l | l.account.availableBalance =
          l.account.availableBalance@pre + earlyTermFee *
          l.account.inventory->one(o|o.realty=realty).fractionOwned) and
7     landlords->forall(l | l.account.realBalance = l.account.realBalance@pre +
          earlyTermFee * l.inventory->one(o|o.realty=realty).fractionOwned)
8 }
```

Figure A.13: Reduce Term

- The term duration of an ACTIVE lease may be reduced by the tenant;

- Reducing the duration of a lease term implies the payment of an early termination fee, defined in the contract terms. The fee is divided between the payees according to the share defined in the contract terms.

```
1 {context Lease::renewTerm(amount:Int)
2   pre: status=COMPLETE and schedule.nextPaymentDue.isAfter(Date.today())
3   post: status=ACTIVE and duration = duration@pre-months + amount
4 }
```

Figure A.14: Renew Term

- A COMPLETE lease may be renewed. This will increase the term duration according to the amount specified.

B

Unit and Integration Test Cases

B.1 Unit Tests

- **Contract Name Service**
  - Set
    - Should return 0x0 for addresses not set
    - Should set the right addresses
  - Update
    - Should update versions when set again
- **MortgageLoan**
  - Deployment
    - Should deploy MortgageLoan
  - Enroll
    - Should enroll on MortgageLoan
    - Should not enroll if wallet transaction fails
    - Should not enroll if already enrolled
    - Should not enroll if not borrower
  - Secure
    - Should secure pending loan (enrolled)
    - Should not secure if not enrolled
    - Should not secure if not lender
    - Should not secure if wallet transaction fails
  - Submit Transaction
    - Should submit and execute transaction immediately on pending loan
    - Should submit but require confirm on active loan

        * Should submit and execute transaction immediately on completed loan
        * Should not submit if not borrower on pending loan
        * Should not submit if not borrower or lender on active loan
    - Amortize
        * Should amortize active loan
        * Should amortize with leftovers
        * Should terminate when fully paid off
        * Should not amortize if already paid off
    - Foreclose
        * Should default if deadline is reached
        * Should not default if not tenant
        * Should not default if deadline is not reached

- **Ownership**
    - Deployment
        * Should set the right owners and shares
    - Approve
        * Should not approve
        * Should transfer shares by internal operator
        * Should transfer shares by external operator
        * Should not transfer shares

- **PaymentSplitter**
    - Deployment
        * Should set the shares right
    - Pay
        * Should divide the payment according to shares

- **RealtyFactory**
    - Deployment
        * Should deploy RealtyFactory contract
    - Mint Asset
        * Should mint asset's Ownership contract
        * Should not mint asset's Ownership contract
    - Add Ownership

        * Should add ownership to new owner
    - Remove Ownership
        * Should remove ownership from previous owner

- **RentalAgreement**
    - Deployment
        * Should set the terms right
    - Enroll
        * Should not enroll if not tenant
        * Should not enroll if transfer fails
        * Should enroll, transferring first month rent and holding security deposit in escrow
        * Should not enroll twice
    - Pay
        * Should not pay if not tenant
        * Should not pay in advance
        * Should not pay more than duration
        * Should conclude after paying everything
        * Should apply the late payment fee
    - Return Deposit
        * Should not return if not complete
        * Should return if complete
        * Should apply penalty
        * Should not let tenant return within return period
        * Should let tenant return after return period
    - Renew Term
        * Should only renew if consent by both parties
        * Should not renew if already terminated
    - Reduce Term
        * Should not reduce if does not respect notice
        * Should reduce and apply fee
    - Terminate
        * Should terminate only if consent by both parties
        * Should not terminate if complete, not enrolled, or security returned
    - Evict

        * Should evict when expired
        * Should not evict if not expired

- **RoleRegistry**
    - Deployment
        * Should deploy RoleRegistry contract
    - Add role
        * Should register new Role
        * Should not register new Role if unauthorized

- **SaleAgreement**
    - Deployment
        * Should deploy SaleAgreement
    - Consent
        * Should not consent if wallet transfer fails
        * Should not consent if ownership transfer fails
        * Should not consent after already consented
        * Should hold assets in escrow when both consented
    - Commit
        * Should not commit if not yet consented by both parties
        * Should not commit if wallet transfer fails
        * Should not commit if already committed
        * Should commit and transfer assets
    - Withdraw
        * Should not withdraw if already committed
        * Should not withdraw if wallet transaction fails
        * Should withdraw and return assets

- **SelfMultisig**
    - Deployment
        * Should set the right participants and shares
    - Submit
        * Should not submit transaction if does not participate
        * Should submit transaction
    - Confirm

        * Should not confirm if does not participate
        * Should execute if majority confirmed
        * Should not execute if not enough confirmations
        * Should not confirm twice
        * Should not confirm if already executed

- **WeightedMultisig**
  - Deployment
    * Should set the right participants and shares
  - Transfer
    * Should transfer shares to other owner
    * Should not transfer shares that doesn't own
  - Submit
    * Should not submit transaction if does not participate
    * Should submit transaction
  - Confirm
    * Should not confirm if does not participate
    * Should execute if majority confirmed
    * Should not execute if not enough confirmations
    * Should not confirm twice
    * Should not confirm if already executed

B.2 Contract Integration Tests

- **Compliance + SaleAgreement + Ownership**
  - Deployment
    * Should deploy
  - AddDocumentation
    * Should add documentation for property kind
  - IsCompliant
    * Should be compliant if no required documents
    * Should not be compliant if required document is not issued
    * Should be compliant if all required documents are issued
  - Integration w/ SaleAgreement
    * Should consent if compliant

        * Should not consent if not compliant

- **MortgageLoan + Wallet**
    - Deployment
        * Should deploy MortgageLoan
    - Enroll
        * Should enroll on MortgageLoan
        * Should not enroll if no wallet allowance
        * Should not enroll if already enrolled
        * Should not enroll if not borrower
    - Secure
        * Should secure pending loan (enrolled)
        * Should not secure if not enrolled
        * Should not secure if not lender
        * Should not secure if no wallet allowance
    - Submit Transaction
        * Should submit and execute transaction immediately on pending loan
        * Should submit but require confirm on active loan
        * Should submit and execute transaction immediately on completed loan
        * Should not submit if not borrower on pending loan
        * Should not submit if not borrower or lender on active loan
    - Amortize
        * Should amortize active loan
        * Should amortize with leftovers
        * Should terminate when fully paid off
        * Should not amortize if already paid off
    - Foreclosure
        * Should default if deadline is reached
        * Should not default if not tenant
        * Should not default if deadline is not reached

- **OrganizationVoter**
    - Deployment
        * Should deploy OrganizationVoter contract
    - SubmitTransaction

        * Should not submit if is not from participating organization
        * Should not submit if is not admin of participating organization
        * Should submit if is admin of participating organization
    - AddParticipant
        * Should not be invoked by anyone except itself
        * Should be invoked if transaction is confirmed
        * Should not add participant if already exists
    - ConfirmTransaction
        * Should confirm, but not execute
        * Should confirm and execute
        * Should not confirm if is not from participating organization
        * Should not confirm if is not admin of participating organization

- **PaymentSplitter + Wallet**
    - Deployment
        * Should set the shares right
    - Pay
        * Should divide the payment according to shares

- **PermissionEndpoints + Permission Registries**
    - Deployment
        * Should deploy PermissionsEndpoints contract
    - Authorization
        * Should not allow anyone to call privileged operations except for OrganizationVoter address
    - Organization Operations
        * Should add organization along with admin role and admin account
        * Should deactivate organization
    - Role Operations
        * Should add role
        * Should not add role of not existing organization
        * Should not add role if has no permission
        * Should not add role with permissions that don't have
    - Account Operations
        * Should add account

        * Should not add account for other organization
        * Should not add account for not existing organization
        * Should not add account for not existing role
        * Should not add account if has no permission
        * Should not add admin if sender is not admin
    - Node Operations
        * Should add node
        * Should not add node for not existing organization
        * Should not add node if has no permission
    - Organization Voter
        * Should not execute operation to add organization
        * Should execute operation to add organization

- **RentalAgreement + Wallet**
    - Deployment
        * Should set the terms right
    - Enroll
        * Should not enroll if not tenant
        * Should not enroll if not enough allowance
        * Should enroll, transferring first month rent and holding security deposit in escrow
        * Should not enroll twice
    - Pay
        * Should not pay if not tenant
        * Should not pay in advance
        * Should not pay more than duration
        * Should conclude after paying everything
        * Should apply the late payment fee
    - Return Deposit
        * Should not return if not complete
        * Should return if complete
        * Should apply penalty
        * Should not let tenant return within return period
        * Should let tenant return after return period
    - Renew Term

        * Should only renew if consent by both parties
        * Should not renew if already terminated
    - Reduce Term
        * Should not reduce if does not respect notice
        * Should reduce and apply fee
    - Terminate
        * Should terminate only if consent by both parties
        * Should not terminate if complete, not enrolled, or security returned
    - Evict
        * Should evict when expired
        * Should not evict if not expired

- **SaleAgreement + Wallet + Ownership**
    - Deployment
        * Should deploy SaleAgreement
    - Consent
        * Should not consent if not approved
        * Should not consent after already consented
        * Should hold assets in escrow when both consented
    - Commit
        * Should not commit if not yet consented by both parties
        * Should not commit if wallet transfer fails
        * Should not commit if already committed
        * Should commit and transfer assets
    - Withdraw
        * Should not withdraw if already committed
        * Should not withdraw if not consented
        * Should withdraw and return assets

C

Application Interface

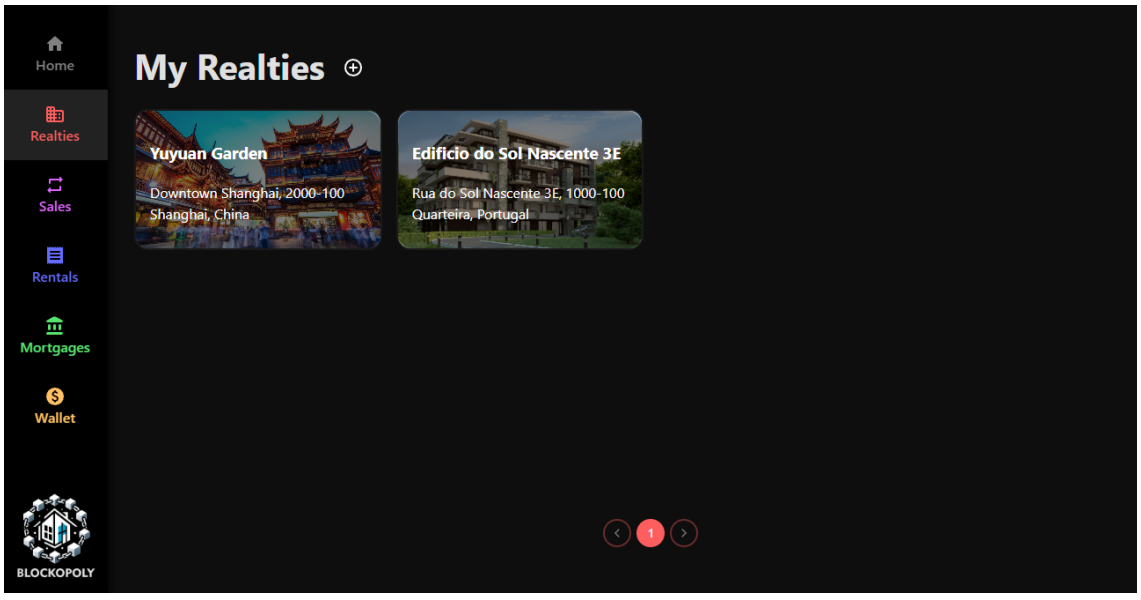


Figure C.1: My Realties Screen

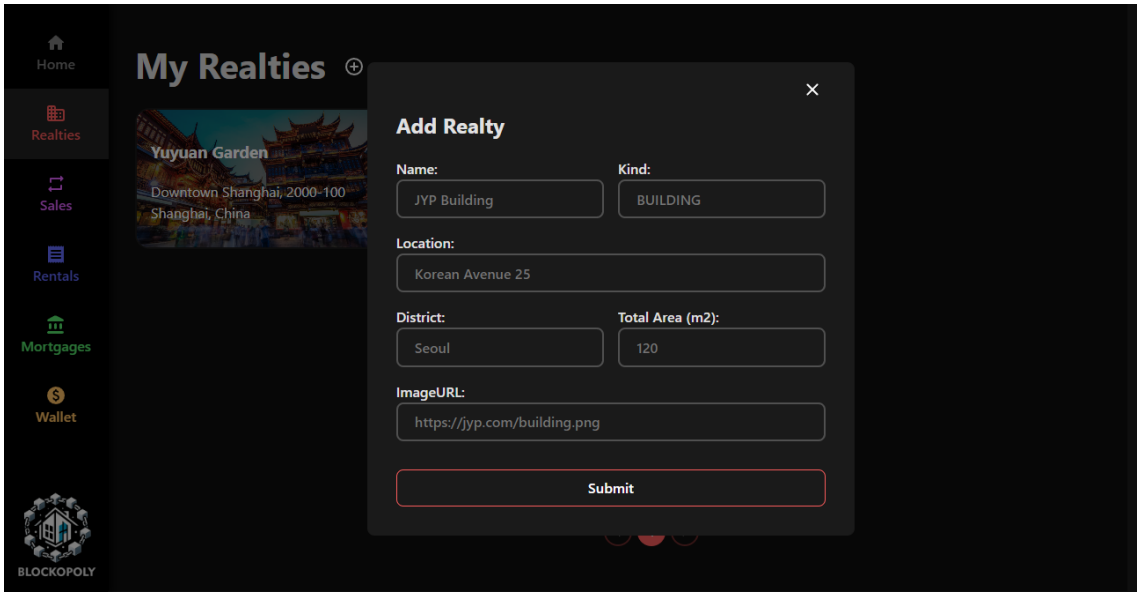


Figure C.2: Add Realty Modal

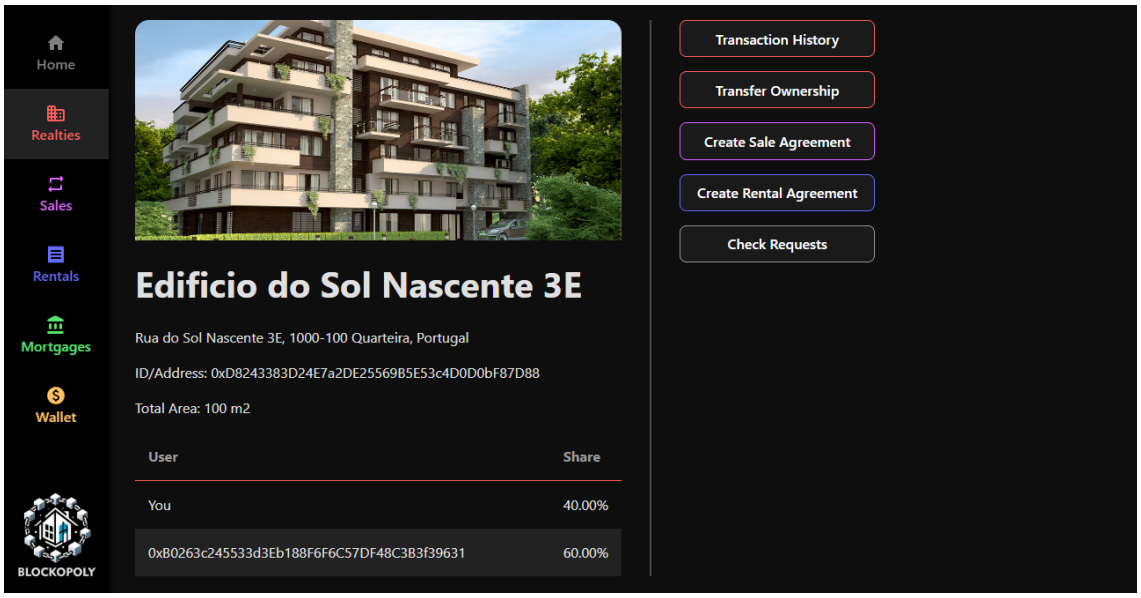


Figure C.3: Realty Screen

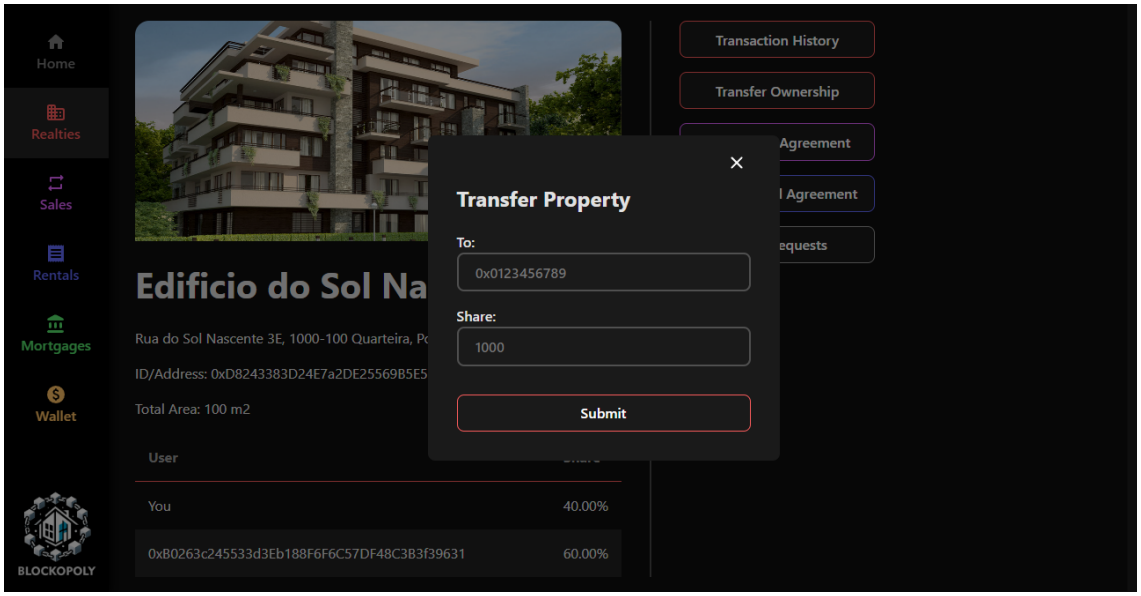


Figure C.4: Transfer Ownership Modal

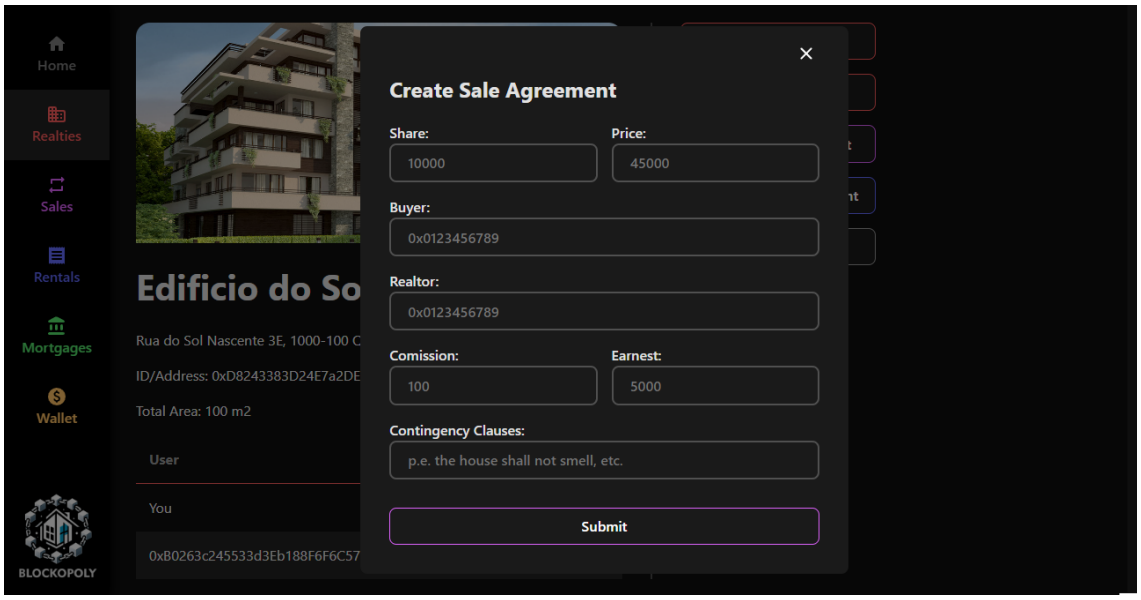


Figure C.5: Create Sale Modal

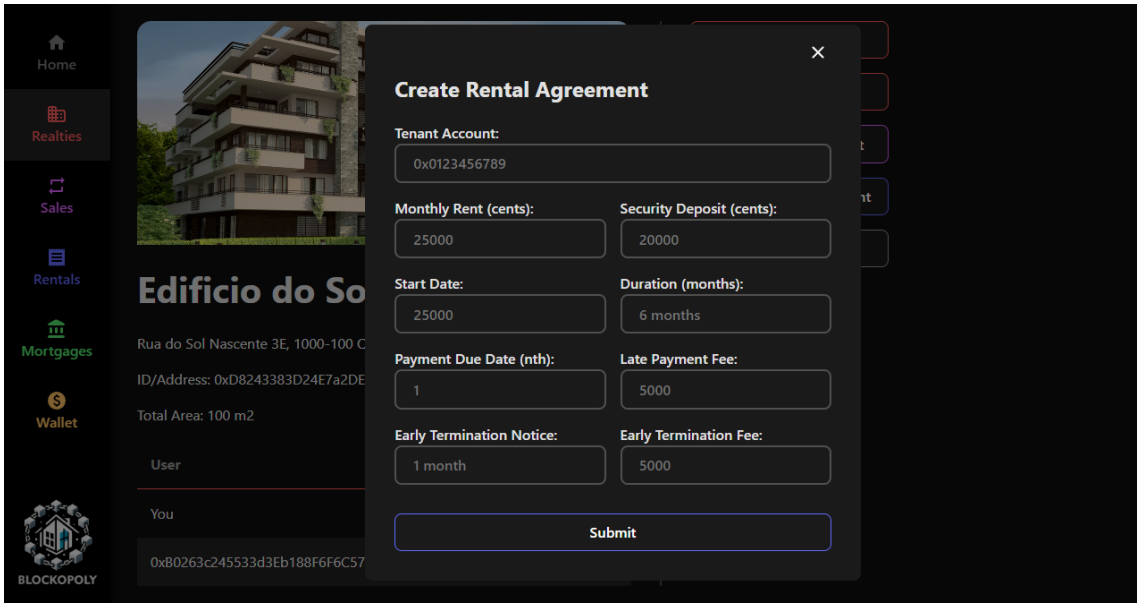


Figure C.6: Create Rental Modal

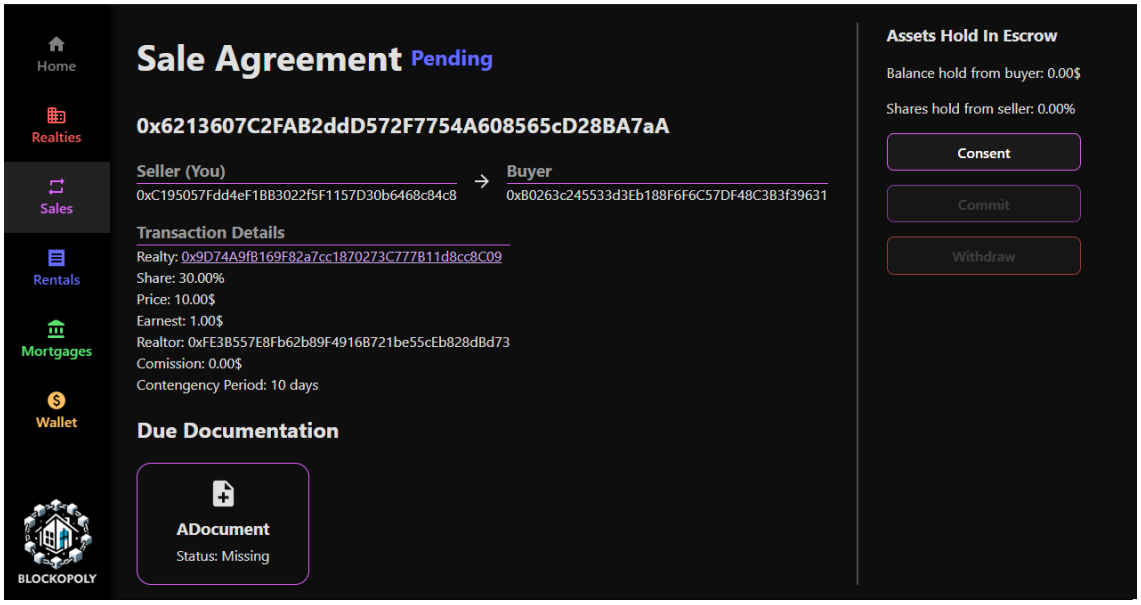


Figure C.7: Sale Screen

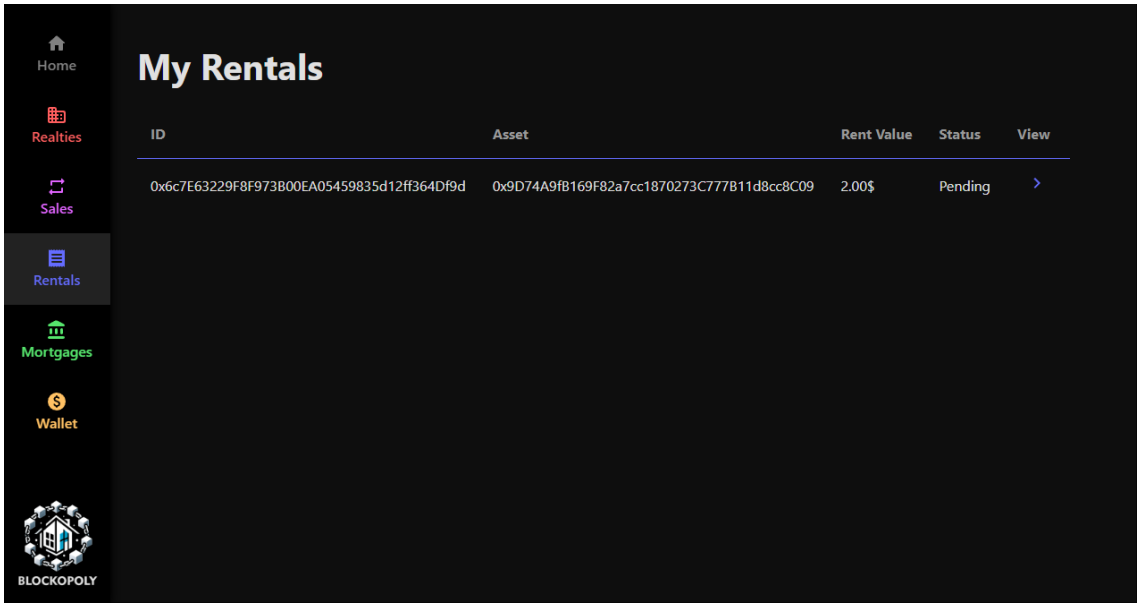


Figure C.8: My Rentals Screen

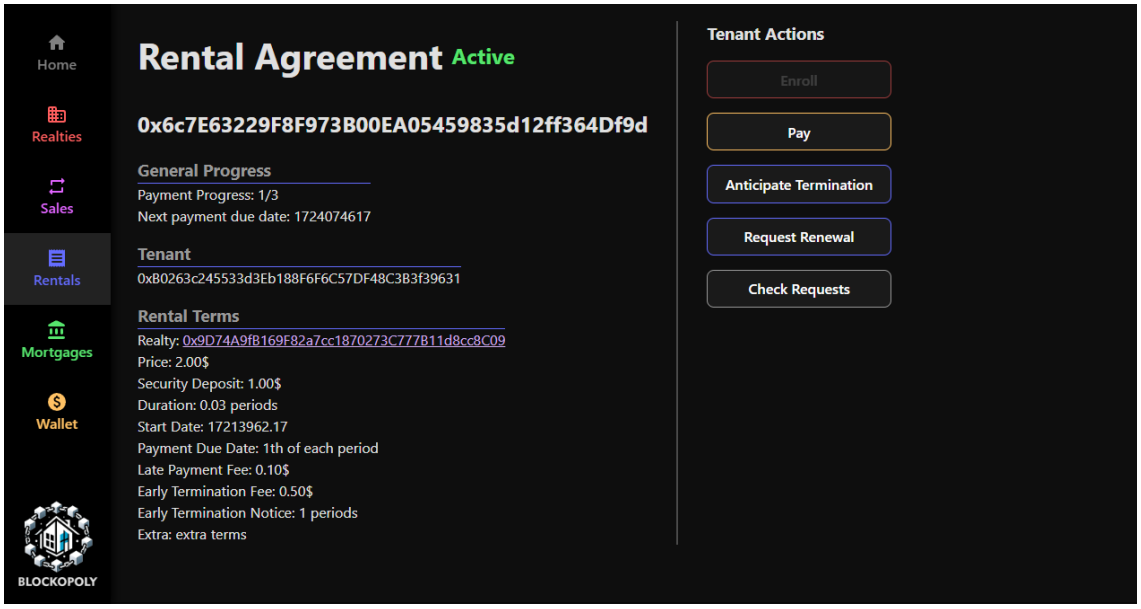


Figure C.9: Rental Screen

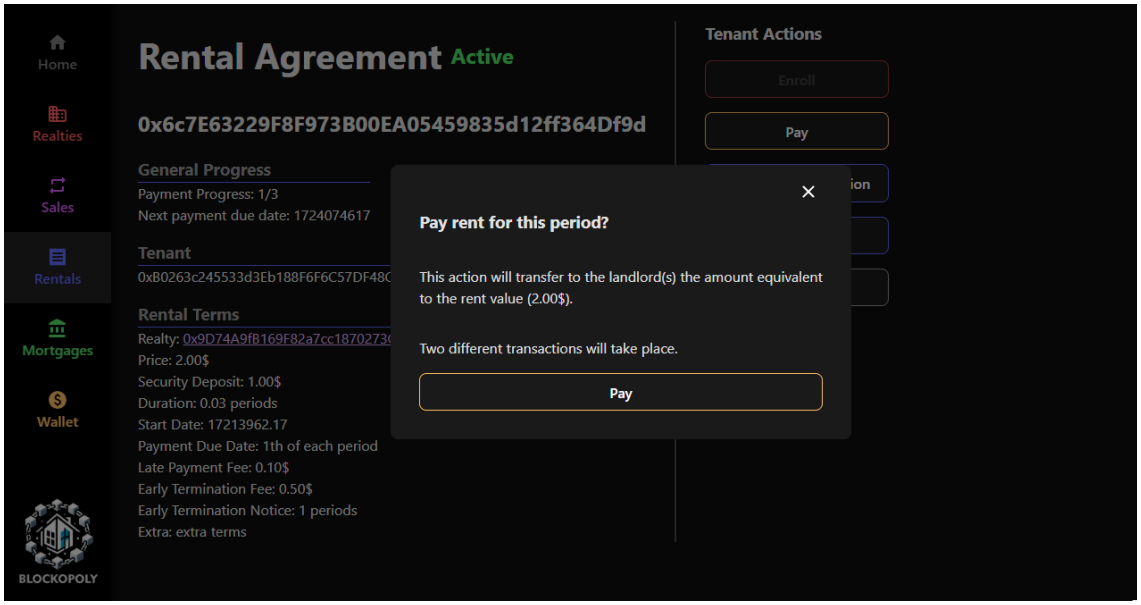


Figure C.10: Pay Rent Modal

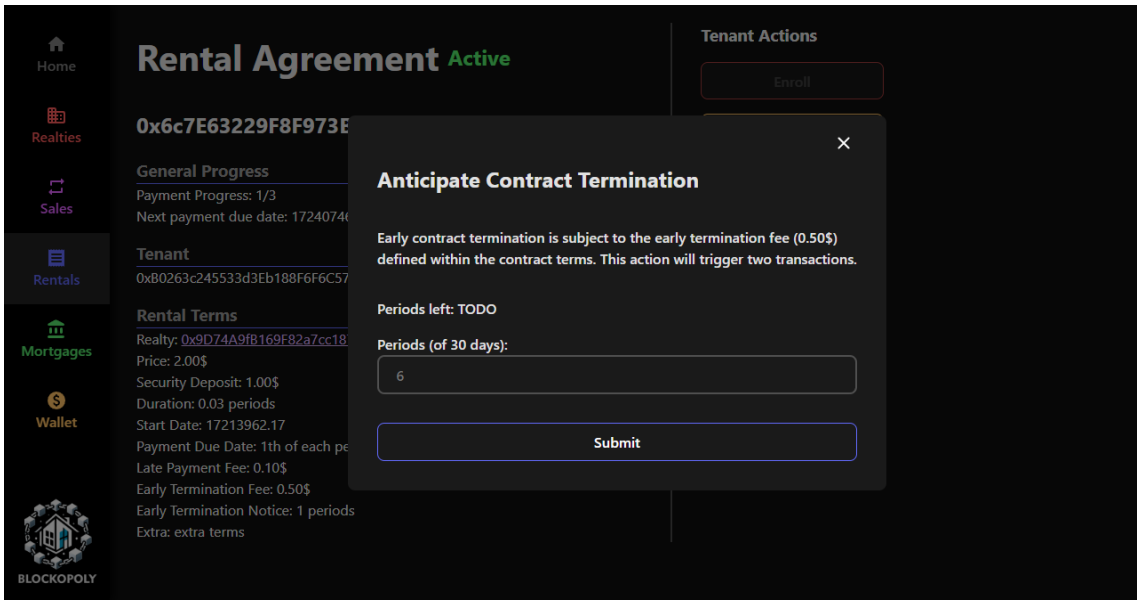


Figure C.11: Reduce Term Modal

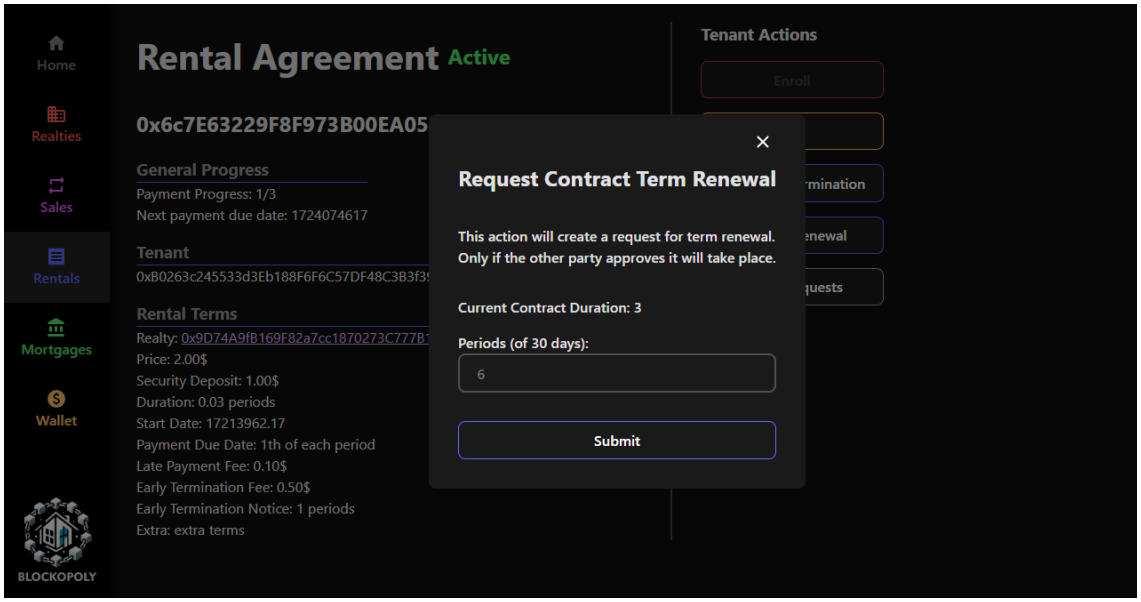


Figure C.12: Renew Term Modal

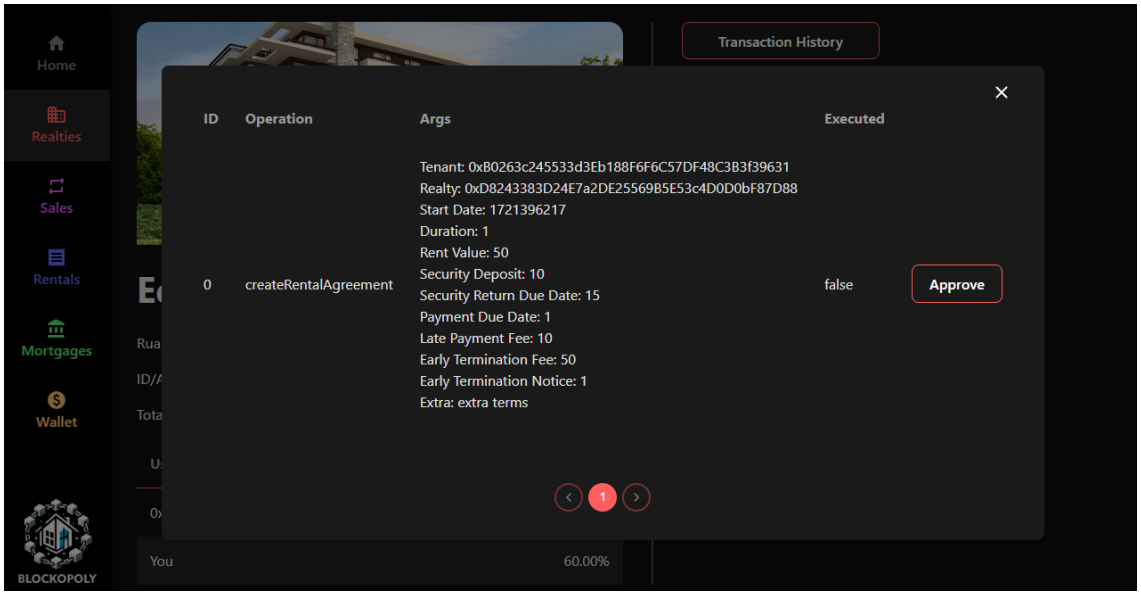


Figure C.13: Requests Modal

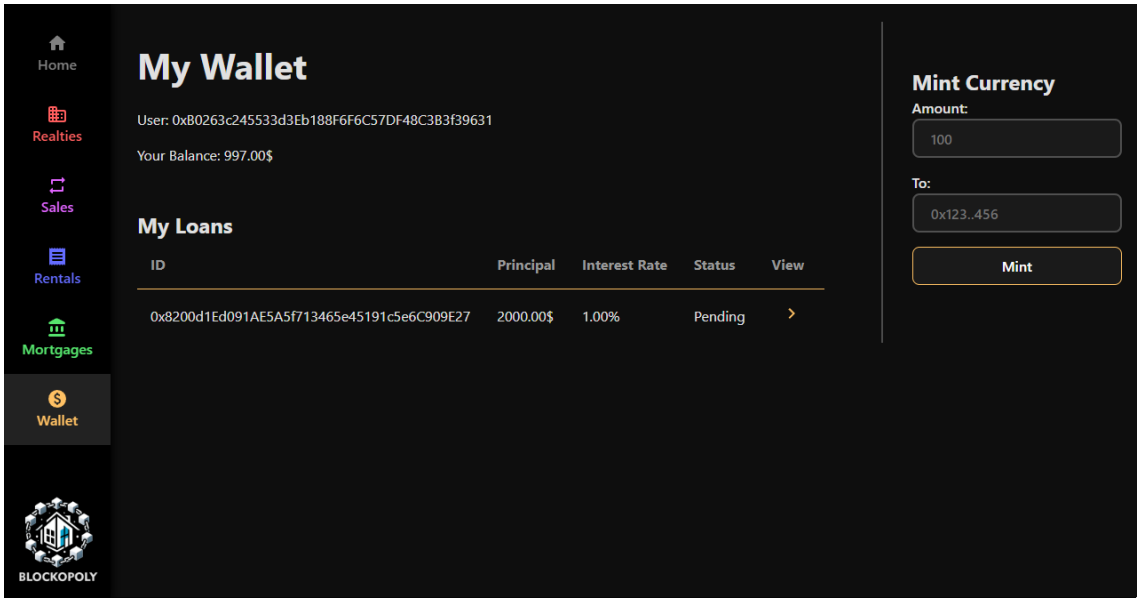


Figure C.14: Wallet