Rodrigo António Marques Quelhas

SECURE AND CONNECT SMART-CONTRACTS TO THE OUTSIDE WORLD

1 2 9 0 UNIVERSIDADE D COIMBRA

1 2 9 0

UNIVERSIDADE D
COIMBRA

Rodrigo António Marques Quelhas

SECURE AND CONNECT SMART-CONTRACTS TO THE OUTSIDE WORLD

Internship report in the context of a Master's in Informatics Security, Specialization in Blockchain Systems, advised by Professor Fernando Boavida and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2021

Faculty of Sciences and Technology

Department of Informatics Engineering

Secure and connect smart-contracts to the outside world

Rodrigo António Marques Quelhas

Internship report in the context of a Master's in Informatics Security, Specialization in Blockchain Systems, advised by Professor Fernando Boavida and presented to the Faculty of Sciences and Technology / Department of Informatics Engineering.

October 2021

UNIVERSIDADE D
COIMBRA

Abstract

Smart contracts are an emerging and promising blockchain technology with a vast set of use cases, which is recently getting more and more attraction due to the rise of decentralized markets and applications. A few examples are decentralized finance, electronic voting, real estate titling records, non-fungible tokens, and supply chain.

The fact that smart contracts run in decentralized systems makes them a viable solution against single entity abuses, which is crucial to prevent corruption.

Unfortunately, a well-known limitation of blockchain systems is the lack of direct access to real-world information (e.g., weather forecasts, stocks, and fiat prices). This constraint makes the technology difficult to use when operations depend on data provisioned by external sources due to the lack of trustfulness with the data providers. Consequently, layer two solutions are necessary to solve the lack of trust and extend the features provided by the bare-metal of blockchain protocols. Layer two is the term used to categorize technologies built on top of the underlying blockchain protocol known as layer one.

The principal focus of the internship was to research and implement secure, reliable, and cost-effective solutions that allow smart contracts to access and use real-world information within the Tezos blockchain. As pointed above, access to external data sources is not supported natively by blockchains, and advancements in this space are necessary for the technology to become more robust and attractive for the less experienced public.

Keywords

blockchain, chainlink, oracle, security, smart-contract, tezos, layer-2

Resumo

Contratos inteligentes são uma tecnologia blockchain promissora com um vasto conjunto de utilidades, sendo que têm recentemente ganho mais atração devido ao crescimento das aplicações e mercados descentralizados. Alguns exemplos reais para uso da tecnologia são o voto eletrónico, registos prediais para real estate, finanças descentralizadas, criação de testemunhos não fungíveis e registo em cadeias de distribuição.

O facto dos contratos inteligentes viverem em sistemas descentralizados faz com que os mesmos sejam uma solução viável no combate a abusos provenientes de entidades singulares, sendo isso crucial na prevenção de corrupção.

Infelizmente, um problema bastante conhecido em sistemas blockchain é a impossibilidade de acesso direto a informações externas, tais como, previsões meteorológicas, preços de ações ou até mesmo rácios de câmbio. Esta limitação faz com que a tecnologia seja difícil de usar com segurança quando as operações dependem de informação provinda de fontes externas devido à necessidade de confiança para com os distribuidores da informação.

Consequentemente, soluções de segunda camada são necessárias para mitigarem a necessidade de confiança e estender as funcionalidades disponibilizadas pelas blockchains. O termo "segunda camada" é usado para categorizar as tecnologias construídas por cima dos protocolos nativos da blockchain, também categorizados como "primeira camada".

O estágio teve como objetivo investigar e implementar soluções seguras, confiáveis e de baixo custo que permitam aos contratos inteligentes acederem e usarem informação disponível no mundo exterior ao sistema blockchain. Como referido acima, o acesso à informação provinda de fontes externas não é suportado nativamente pelas blockchains, por isso, avanços neste espaço são necessários para que a tecnologia se torne mais robusta e atrativa ao público menos experiente.

Palavras-Chave

blockchain, chainlink, oráculo, segurança, smart-contract, tezos, layer-2

Contents

Acronyms

API Application Programming Interface. 14

BFS Breadth-first search. 10

CBDC Central Bank Digital Currency. 61

DAO Decentralized Autonomous Organization. 29

DeFi Decentralized Finance. 14

DFS Depth-first search. 10

DPoS Delegated Proof of Stake. 6, 11

DSL Domain Specific Language. 7, 9

eDSL Embedded Domain Specific Languages. 7

EVM Ethereum Virtual Machine. 11, 54

LPoS Liquid Proof of Stake. 1, 6

NAPR National Agency of Public Registry. 16

NFT Non-Fungible Token. 14

NPoS Nominated Proof of Stake. 11

PoS Proof of Stake. 1, 6, 11

PoW Proof of Work. 1, 11

PRE Polkadot Runtime Environment. 11

TDD Test Driven Development. 56

TTL Time to Live. 28, 29, 41, 43

VRF Verifiable Random Function. 14

List of Figures

List of Tables

Listings

Chapter 1

Introduction

1.1 Scope

This document reports the work done in an internship whose principal focus was researching and implementing secure, reliable, and cost-effective solutions that allow smart contracts to access and use information from the real world within the Tezos blockchain.

Smart contracts are digital protocols intended to facilitate, perform, verify and enforce the negotiation and performance of agreements or tasks without the need for trust and third parties [33].

Tezos is a three-year-old blockchain that uses a variant of Proof of Stake (PoS) as a consensus algorithm, called Liquid Proof of Stake (LPoS) [14]. PoS was officially introduced [1] in 2012 with the purpose of solving the high energy consumption problem existent with Bitcoin mining [20]. When comparing PoS with Proof of Work (PoW), the principal difference is that PoW relies on the users computational power, where PoS relies on the amount of security deposits users have at stake [15].

The internship occurred during my full-time consulting activities for Smart Chain Arena, a company based in the United States. Our focus is on developing tools and infrastructure for Tezos, including providing technical knowledge to the community, where one of those tools is **SmartPy**, an embedded domain-specific language and platform for writing contracts. The platform currently offers three dialects, **Python**, **Typescript**, and **Ocaml**, where the Typescript and Ocaml dialects are still recent and considered experimental.

[1] The original idea comes from a forum named bitcointalk.org in 2011, where a user proposed the Proof of Stake concept for the first time. https://bitcointalk.org/index.php?topic=27787.msg349645

At SmartPy, we received a grant from Chainlink to provide Tezos developers with several key Chainlink functionalities to build more advanced and secure smart contracts applications. Chainlink is a set of protocols and tools that enable smart contracts on any blockchain to leverage off-chain resources, such as verifiable randomness, tamper-proof price data, external APIs, and various other things [24]. I found the topic interesting, which led me to choose it as a base for the internship's proposal.

The information about the Chainlink grant is available at:
https://blog.chain.link/smartpy-receives-grant-to-integrate-chainlink-price-feeds-on-tezos

The core idea is to increase decentralization by having pools of oracles provisioning and aggregating data on-demand or between specific time intervals. An oracle is a bridging mechanism that interfaces smart contracts with off-chain data sources [4] as illustrated in figure 1.1.

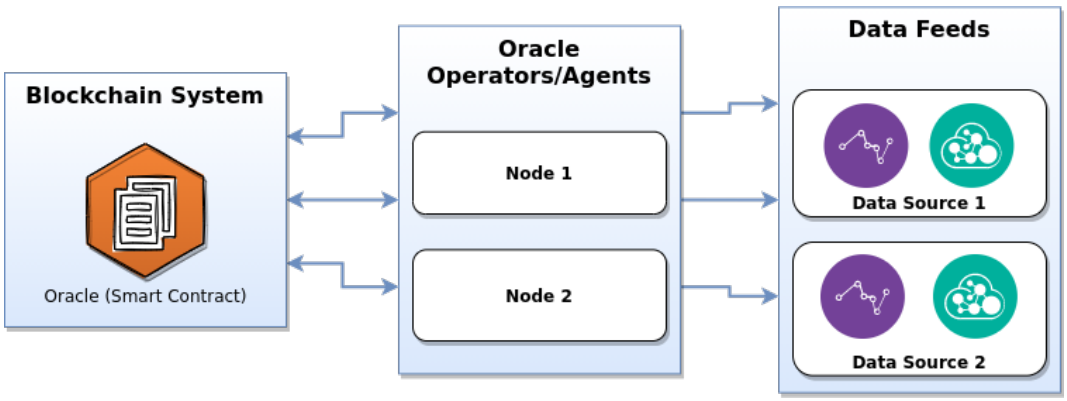

Figure 1.1: Oracles in a blockchain system.
Oracle smart contracts allow the interconnection between other contracts and off-chain data sources.

For the solutions to be secure, reliable, and cost-effective, this work looked into approaches to prevent the loss of funds and provision of false information while still allowing for low gas and storage consumption, which result in cheap transaction fees compared with other blockchains like Ethereum.

1.2 Motivation

I have been involved with blockchain technologies for almost three years, including developing minting software and a meta-programming language and platform called SmartPy for writing smart contracts, which is currently the most used language in the Tezos blockchain.

Also, I find blockchain technologies interesting because of their decentralized nature, which in my opinion, is becoming more relevant and pivotal in this new digital world.

1.3 Objectives

Apart from what was described in the scope of this chapter, the principal objective was to provide solutions on the Tezos blockchain to enable users to connect their smart contracts with external data sources by using state-of-the-art technology provided by the Chainlink ecosystem.

It also includes adding and adapting new features into SmartPy that contribute to the language and platform to become more robust, easy to use, and feature-complete.

1.4 Report Structure

This section describes the report structure. It is composed of four main chapters that contain the ultimate conclusions of the internship.

1. **Introduction** – Introduces the reader to the subject of the report, the motivation behind it, the scope, the objectives, and a compact overview of the report organization;
2. **State-of-the-Art and Background** – Aims to show and explain best practices and technologies that can be used to improve the security and reliability of smart contracts when accessing off-chain information;
3. **Security Considerations** – Presents various security considerations taken into account during the development of the solution;
4. **Requirements and Design** – This chapter presents the requirements and high-level design of the solution.;
5. **Implementation Overview** – A brief overview of the implementation process and the technologies used;
6. **Testing** – This section details the testing procedures taken to ensure and validate the quality of the solution;
7. **Conclusions and Future Work** – Presents the most important conclusions of the internship and adds suggestions of features that could be implemented to improve the project's overall workflow, making it more complete and user-friendly.

Chapter 2

State-of-the-Art and Background

2.1 Introduction

This chapter focuses on some of the best practices and technologies currently being used in blockchain systems, mainly on their capability to improve the security and reliability of smart contracts.

- Section [2.2] describes why the Tezos blockchain is an excellent platform for smart-contract applications;
- Section [2.3] presents some of the current issues when using smart contracts in real-world applications and proposes viable solutions for those problems.
- Section [2.4] shows some of the technologies and practices currently in use to connect smart contracts to the outside world;
- Section [2.5] presents examples of applications for smart contracts, where the technology would bring benefits;
- Section [2.6] includes the main conclusions of this chapter.

2.2 State-of-the-Art on Tezos

The popularization of Bitcoin, a decentralized cryptocurrency, has inspired the creation of several alternative solutions that bring their own source of innovation. Cardano, Ethereum, Monero, and Zcash are good examples since they all represent unique contributions to the cryptocurrency space. However, most blockchains have no means of adopting the advancements from other cryptocurrencies without

a strong possibility of causing hard-forks [34]. Tezos was built with this problem in mind and includes a self-amending mechanism that facilitates protocol upgrades by leveraging the consensus algorithm to give the right to its validators to elect a new protocol in the blockchain [1].

2.2.1 What problems does Tezos aim to solve?

As mentioned above, Tezos attempts to remedy the upgradeability problem in the cryptocurrency space by having a self-amending mechanism that facilitates protocol upgrades through an election process. It gives its validators the power to vote if a new protocol should be adopted or not, reducing the need for hard-forks significantly [1]. New protocols can even amend the self-amending mechanism itself and include development rewards that get distributed once the protocol is adopted.

Tezos was one of the first blockchains to implement the Liquid Proof of Stake (LPoS) consensus algorithm, a variant of Proof of Stake (PoS) designed to solve the high energy consumption problem present in blockchains like Bitcoin and Ethereum [20]. LPoS extends PoS by allowing every wallet to delegate their balance to a block producer/validator, where each wallet can explicitly give staking rights to their bond without transferring the ownership to the tokens [2].

In figure 2.1 is possible to see the differences between LPoS and Delegated Proof of Stake (DPoS), which are both variants of Proof of Stake.

	Liquid Proof-of-Stake (Tezos)	**Delegated Proof-of-Stake (EOS, Lisk, TRON, BitShares)**
Delegation (Purpose)	❖ Optional (minimizes dilution of small token holders)	❖ Required to elect block producers (enables greater scalability)
Barrier to Entry	❖ 8,000* ꜩ, with 8.25%** of total baked tokens (including delegated) frozen as bond ❖ Modest computing power & reliable internet connection	❖ Professionalized operations with significant computing infrastructure ❖ Competition from other delegates
Validator Set	❖ Dynamic (size not fixed) ❖ Up to 80,000 bakers*** (limited by roll* size)	❖ Fixed size ❖ 21 in EOS, 101 in Lisk, 27 in TRON
Design Priorities	❖ Decentralization, accountable governance, and security	❖ Scalability and usable consumer applications

* ~$30,000 at time of writing. The protocol makes random selection of a stakeholder more efficient by selecting randomly from rolls of 8,000 ꜩ instead of atomic units of tez. This amount can be reduced (e.g. to 2000 ꜩ) over time via the amendment process.
** Bond size will ramp up from 0% to 8.25% over 6 months to bootstrap the network.
*** Anywhere near this upper limit is extremely unlikely, but it reflects the order of magnitude difference in the number of validators possible in Tezos.

Figure 2.1: Liquid Proof of Stake on Tezos.
A comparison between LPoS and DPoS consensus algorithms.
Source [2]

2.2.2 Smart Contracts & Formal Verification

Smart contracts on Tezos are expressed in a stack-based programming language called **Michelson**, the native language understood by Tezos interpreter, which was developed explicitly for Tezos. It contains strict type-checking and high-level data types and primitives. Furthermore, Michelson was designed with formal verification in mind, a method used to improve the security and testability of smart contracts by mathematically proving the properties of programs [5].

Developing smart contracts on Tezos is straightforward, thanks to the high-level programming languages available in the ecosystem, giving many options to developers wanting to write applications on top of smart contracts. In addition, some of those languages are Embedded Domain Specific Languages (eDSL)'s hosted in well-known programming languages like Python, Haskell, and Ocaml, facilitating the learning curve for those already accustomed to these syntaxes. An embedded Domain Specific Language (DSL) inherits the features provided by its host language, like controls, standard libs, expressions–and adds domain-specific abstractions on top to facilitate the programmer's work.

2.2.2.1 High level languages:

Disclaimer: It is important to note that I may be biased towards SmartPy in the advantages and disadvantages mentioned below, mainly due to being one of the maintainers of SmartPy.

- **SmartPy**[1] - An embedded Domain Specific Language eDSL, which supports three dialects, Python, Typescript and Ocaml;

  Pros:

  - Has three dialects: Python, Typescript and Ocaml;
  - Offers a powerful test framework that users can use to unit test most of the functionalities;
  - Provides a great variety of templates that users can use to learn or bootstrap their projects;
  - Supports the majority of the features allowed by Michelson;
  - Has an online IDE that supports all three dialects;
  - Offers meta-programming capabilities, which is helpful for creating configurable contracts.

[1] https://smartpy.io

- Short learning curve;

Cons:

- In the Python dialect, variables have to be wrapped in a local construct, which can cause some confusion to the users;

Below is a code snippet 2.1 that shows a minimal example of a smart contract written in SmartPy.

```
import smartpy as sp

class MinimalContract(sp.Contract):
    def __init__(self):
        self.init(1)

    @sp.entry_point
    def updateStorage(self, newStorage):
        self.data = newStorage
```

Listing 2.1: A minimal smart contract example written in SmartPy.

- **Ligo**[2] - A polyglot language with various syntaxes, like Ocaml, Pascal, ReasonML and Javascript;

Pros:

- Provides four dialects: Ocaml, Pascal, ReasonML and Javascript;
- Supports the majority of the features allowed by Michelson;
- Good integration with formal verification tools;
- Has an online IDE;

Cons:

- The test framework is still very limited;
- Long learning curve for non-function developers;

Below is a code snippet 2.2 that shows a minimal example of a smart contract written in Cameligo.

```
type storage = nat

type parameter =
  Update_storage of nat
```

[2]https://ligolang.org

```
type return = operation list * storage

let update_storage (new_value: nat) : storage = new_value

let main (action, store : parameter * storage) : return =
 ([] : operation list),
 (match action with
   Update_storage (n) -> update_storage (n))
```

Listing 2.2: A minimal smart contract example written in Cameligo.

- **Archetype**[3] - A DSL which facilitates formal verification;

  Pros:

  - Good integration with formal verification tools;
  - Can transpile to **Ligo**;
  - It integrates well with VS Code.

  Cons:

  - Doesn't have a test framework;
  - Its syntax can be difficult to understand;

  Below is a code snippet 2.3 that shows a minimal example of a smart contract written in Archetype.

```
archetype minimal

variable value : int = 1

entry main (new_value : int) {
  effect {
    value := new_value
  }
}
```

Listing 2.3: A minimal smart contract example written in Archetype.

2.2.3 Why using Tezos for developing smart-contracts?

As previously mentioned, I have been working with the Tezos blockchain for almost three years now, including being one of the maintainers of a high-level smart contract

[3] https://archetype-lang.org

programming language and platform called **SmartPy**. I find the self-amending mechanism a significant advantage since it allows the blockchain protocol to upgrade every three months through an election process, where the community has the power to decide if the protocol should be adopted or not.

2.2.3.1 Why do protocol upgrades usually take three months?

In theory, the duration of a protocol proposal is two and half months, divided into five phases of fourteen days[4]: **Proposal**, **Exploration**, **Cooldown**, **Promotion**, and **Adoption**. Furthermore, proposals take between three and four months because they require preparation and testing before getting injected for election on-chain.

The latest upgrades to the Tezos blockchain provided many improvements to the Michelson language. Those improvements include:

- Gas cost reductions which resulted in cheaper transactions;
- Increased storage and gas limits, doubling the maximum allowed size for smart contracts;
- Changed the execution order of operations from Breadth-first search (BFS) to Depth-first search (DFS) solving a variety of potential runtime vulnerabilities as described in section 2.3.1;
- Support for multiple big maps containing huge amounts of data accessed only on-demand resulting in fewer deserialization costs.

Michelson was also designed to facilitate formal verification [6]. Formal verification is a considerable advantage since it allows the contracts to be proved correct mathematically against a specification.

Fee costs are a huge consideration when choosing the platform for smart contracts, and Tezos is currently one of the cheapest.

Table 2.1 provides a list containing some of the platforms for smart contracts, which in my opinion, are the most promising. I consider Polkadot parachains the best alternative to Tezos since it is a relay chain that improves interoperability between multiple chains, which has enormous potential.

[4] https://www.tezosagora.org/learnthe-five-stages-of-tezos-governance
[5] IELE is a variant of LLVM (a register-based machine)
[6] Polkadot doesn't support smart-contracts natively, only its parachains support it.

Blockchain	Runtime	Consensus	Native Language
Tezos	Tezos VM	DPoS	Michelson
Ethereum	EVM	PoW, moving to PoS	EVM Code
Cardano	KEVM, IELE [5]	PoS	KEVM code, IELE code
Polkadot	PRE	NPoS	WebAssembly[6], ...

Table 2.1: Smart Contract Platforms

2.3 Known problems in smart-contracts

2.3.1 Reentrancy Vulnerabilities

Reentrancy is a vulnerability that usually affects smart contracts. The vulnerability exists when a contract calls another contract without first updating the storage state, allowing malicious contracts to take advantage of the faulty state by calling back the original contract as many times as they deem necessary, resulting in the loss of funds in most cases [12].

A well-known exploit to this vulnerability happened on Ethereum, the so-called **The DAO Hack**[7] and resulted in a hard-fork which originated Ethereum Classic.

2.3.2 Scaling problems and access to real-world data

Blockchains are secure systems that achieve data security and integrity by having a distributed and auditable storage validated by multiple entities through the consensus algorithm that enforces rules for all participants. It makes the scalability of the technology very difficult, and in most cases, the supported amount of operations per second is meager compared to technologies like Paypal or Visa.

Furthermore, some consensus algorithms like the one used in **Avalanche**, a layer one PoS blockchain, can achieve quick finality and high throughput, allowing thousands of transactions per second [28]. Tezos is working on adding deterministic finality, which improves the block finality from a minimum of 15 minutes to 30 seconds, where the state of the blockchain becomes irreversible after two blocks [3]. Consequently, smart contract applications will become quicker, which should attract more users to the ecosystem.

Layer two solutions are also great to improve the scalability and access to real-world data in blockchain systems. The term "Layer two" is used to categorize technologies

[7] https://www.gemini.com/cryptopedia/the-dao-hack-makerdao

built on top of the underlying blockchain protocol known as "Layer one".

2.3.2.1 What are layer two solutions?

Layer two solutions are protocols or frameworks built on top of the underlying blockchain technology by extending its features and use cases. The principal objectives of these frameworks are usually to improve scalability limitations by adding high throughput, reducing cost fees, bridging multiple blockchains, and facilitating interoperability with off-chain applications [17]. The majority of consensus algorithms currently being used by public blockchains cannot yet compete with standard payment solutions, which is undoubtedly a significant limitation presently blocking mass adoption of the technology.

2.3.2.1.1 State Channels

Figure 2.2 illustrates a kind of layer two solution, usually called state channels. The objective of state channels is to increase operations throughput and reduce fees by aggregating multiple off-chain commitments into a few on-chain operations, resulting in fewer on-chain calls while keeping security and data integrity.

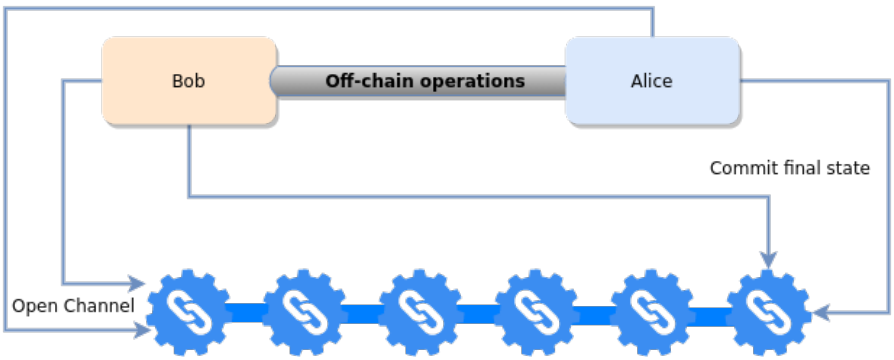

Figure 2.2: An illustration of how state channels work.

Good examples of state channel solutions are the **Lightning network**[8] on bitcoin, **Plasma**[9] and **ZK-Rollups**[10] on ethereum [27].

2.3.2.1.2 Oracles

Another layer two solution is oracle services, which consist of external agents connecting smart contracts to the outside world. Those oracles can be used as single

[8] https://lightning.network
[9] https://ethereum.org/en/developers/docs/scaling/plasma
[10] https://docs.ethhub.io/ethereum-roadmap/layer-2-scaling/zk-rollups

agents that simply provide information on-demand into the smart contract or as distributed networks that use on-chain mechanisms to reach consensus between all the participants when submitting new information [4].

Figure 2.3 illustrates the core concept of an oracle, where a smart contract can request and receive off-chain data from external API's through the use of a middleware agent. This simple approach has a few security concerns, as I will explain below in section 2.4.1.

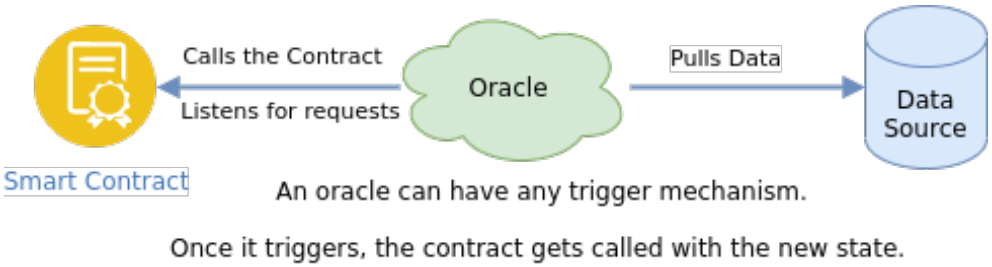

Figure 2.3: A simple illustration of how oracles work.

2.4 State-of-the-Art in Oracle Solutions

Oracles are data sources/providers used as bridge mechanisms between smart contracts and the outside world. Furthermore, an oracle or a group of oracles are external agents connected to the smart contract and off-chain data sources. They can verify and authenticate the authenticity of the data being provided through consensus, and they are vital for providing reliable information into smart contracts.

2.4.1 The Oracle Problem

The principle behind oracles goes against the main objective of blockchain systems, which is to have a fully decentralized ecosystem where every interaction is trustless. In most situations, oracles are implemented as centralized applications and have tremendous power over smart contracts since they can mutate their state [11]. Furthermore, being centralized opens the possibility for many vulnerabilities because it only has a single source of truth. If the oracle gets controlled by a malicious actor, the resulting consequences can be catastrophic. However, consensus oracles can solve this problem since they are designed similarly to blockchains and can be secure, reliable, and trustless by applying decentralized aggregation of data.

2.4.2 Oracle Solutions

2.4.2.1 Chainlink

Chainlink is one of the most advanced oracle systems currently available [26]. Its first white-paper [30] was released in September of 2017, and its implementation launched in May of 2019. The technology was initially built on top of the Ethereum network and is now available in multiple blockchains. It is supported by a big community and is successfully being used by many applications, like Decentralized Finance (DeFi), external payments, gaming (Non-Fungible Token (NFT)s and Randomness), supply chain, and utilities.

Chainlink is composed of various frameworks, protocols, and a digital token. The digital token is called **Link** and is used as a currency in the Chainlink ecosystem. It serves as a business model providing incentives for a healthy ecosystem, where participants get incentivized to participate by offering services in exchange for payments with **Link**.

What solutions does chainlink protocol provide?

These are a few solutions extracted from the information available on the Chainlink website[11].

- **Price Feeds** - Chainlink provides decentralized price feeds, which are aggregated and validated on-chain by multiple trusted entities. Those price feeds contain pairs of commodities prices like precious metals, fiat currencies, and cryptocurrencies.
- **Verifiable Random Numbers** - Verifiable Random Function (VRF) is a verifiable source of randomness and is often used for:
  - Games built on top of smart contracts;
  - Generating NFTs with random properties that make them valuable;
  - Consensus mechanisms implemented on smart-contracts;
- **Access to external APIs** - These Application Programming Interface (API)s allow interoperability between blockchains and outside data sources.

Recently, chainlink released two new white papers [24] and [23], which bring significant improvements to the underlying solutions by adopting a consensus mechanism

[11] https://docs.chain.link

for off-chain reporting that will result in a more efficient approach for distributed oracles. It suggests various algorithms, which consist of aggregating, signing, and broadcasting information off-chain between multiple nodes in a peer-to-peer network, including the random selection of responsive leaders to inject the operations into the chain.

2.5 Smart contract applications

2.5.1 Land Registry

Land titling usually involves many steps, is a cumbersome process, typically too fragmented, and manual processes can lead to document manipulation and, consequently, fraud. The process requires various entities to examine reasonable amounts of documents regarding the agreements, construction history, land past owners and mortgage confirmations [29].

In figure 2.4 we can see the entities and the workflow differences between the traditional system and the one leveraging smart contract technologies.

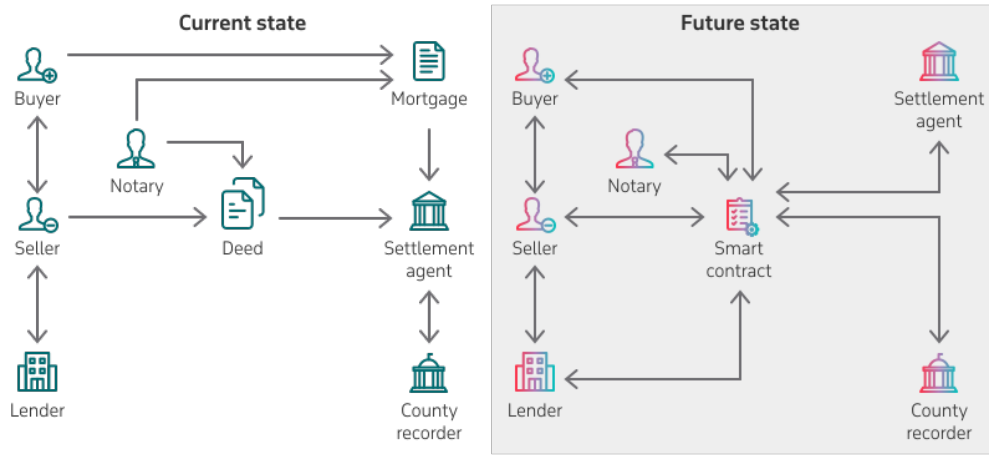

Figure 2.4: Usage of smart contracts for land titling.
Source: [21]

Smart contracts leverage the cryptographic primitives available in blockchains to keep a trusted, distributed, and continuous history of all land registrations, securing and facilitating the validation process when changing the land ownership. It reduces third-party dependencies and, as a result, could reduce the total cost of land registrations.

2.5.1.1 Adoptions in the real world

- Securitize and Elevated Returns are digital asset management firms focused on decentralized finance in digital asset securities. The quotes bellow refers to their intention to move one Billion dollars worth in real estate assets to Tezos smart contracts from their initial deployments on the Ethereum blockchain.

  > *"Securitize and Elevated Returns to Tokenize $1 Billion of Real Estate Assets on Tezos"*
  >
  > *"Having worked closely with regulators and local authorities around the world, we understand the need for the highest security and compliance features. There is no better solution than working on a Tezos-based token implementation. We have a number of very high-profile deals lined up and we could not afford to compromise the technological product. With the Tezos-powered solution and its integration in the Securitize portal technology, we feel we have a total solution."*
  > **Source: [13]**

- Another use case in the Tezos blockchain, Vertalo, a digital transfer agent, is helping to tokenize a Pennsylvania-based real estate portfolio of class A properties.

  > *"Vertalo, tZERO Are Bringing $300M in Real Estate to the Tezos Blockchain"*
  > **Source: [9]**

- A Blockchain-Based Land Titling project in the Republic of Georgia

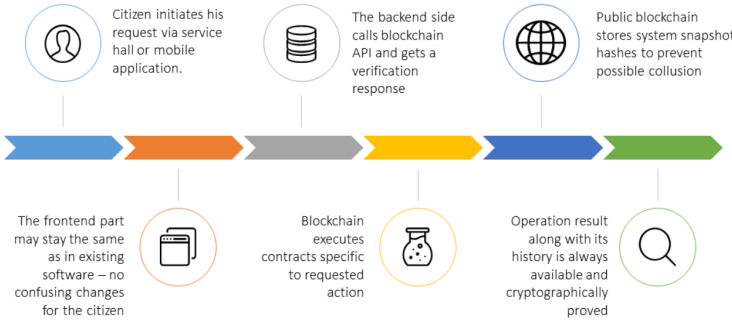

Figure 2.5: Georgian land titling workflow. **Source: [16]**

Republic of Georgia's National Agency of Public Registry (NAPR) adopted the first-ever blockchain used for land registry. It was developed by Bitfury

and launched in 2016, aiming to secure property ownership rights, facilitating registration processes, and increasing trust in the government. Since then, the pilot has been a success, and citizens now use digital certificates to prove their land ownership, with over 1.5 million land titles already registered[29].

2.5.2 Escrow Services

Smart contracts are a good addition to escrow services. They can automate the process, ensuring trust and removing intermediate third parties, which may result in cost reductions [31].

Applications for escrow smart contracts:

- **Decentralized freelance platforms** - Smart contracts are also an excellent solution for freelancing service payments. Task and job postings can be submitted on a platform running on top of a smart contract that holds the service payments while the work is not finished, removing the need for intermediaries. Examples of platforms of this kind are **cryptotask**[12] and **SmartLink**[13] on Tezos.
- **Electronic auctions** - Sellers would put their items for sale in an "Auction smart-contract" where bidders bid on items by depositing the respective amounts and refreshing the highest bid. As illustrated in figure 2.6, the bidding process could be automated by a smart contract, leaving only the pre-sale and item validations to be managed by an off-chain entity where oracles could be used to update item status and initiate the auction of items. One example of a platform of this kind is **tzcolors**[14], an NFT auction experiment on Tezos.

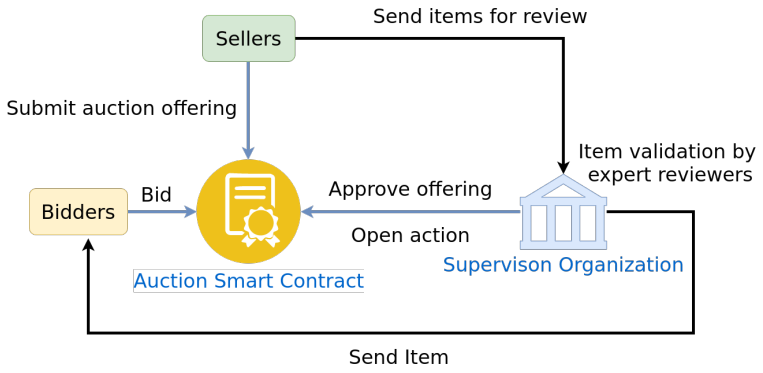

Figure 2.6: An illustration of electronic auction using smart contracts.

[12]https://app.cryptotask.org/en/freelancers
[13]https://www.smartlink.so/products
[14]https://www.tzcolors.io

2.6 Conclusion

This chapter presented relevant features currently being used in blockchain systems, the concepts, and the ideas that make them promising for the future of smart contracts.

It also focused on why the Tezos blockchain is an excellent platform for hosting secure, reliable, and cost-effective smart contracts. No matter what innovations other blockchains produce, it will be possible for Tezos validators to adopt these innovations without a high risk of causing hard-forks.

Chapter 3

Security Considerations

3.1 Introduction

The **Michelson** language was designed with the purpose to make smart contracts more secure and easy to write. Although writing vulnerable ones is still possible, developers should always think deeply about the security issues that can occur by looking at the history of previous vulnerabilities, understanding them, and finding solutions to prevent them from happening. In addition, various kinds of vulnerabilities exist that primarily affect smart contracts. This section studies some of those vulnerabilities, which were taken into account when designing the solution specification.

3.2 Replay Attacks

Replay attacks are a recurrent issue in smart contracts and almost always happen because the developer did not understand well enough how signatures work.

3.2.1 What makes this attack possible?

It is pretty simple. Suppose that the same contents to be signed can occur multiple times. In that case, it means that a malicious actor could record all previous signatures and wait for a chance where the contents will be identical to allow him to bypass the signature validation and impersonate the original wallet.

3.2.2 How can it be prevented?

The implementation must ensure that the contents to be signed cannot happen more than once. There are many solutions for this problem, like appending a random nonce to the contents before signing it. The multi-signature contract specification has been designed with this in mind.

3.3 Reentrancy Attacks

Reentrancy attacks are probably the most known since one of those caused a hard-fork on **Ethereum** after the famous **the DAO Hack**[1] where millions of dollars got stolen, causing the network to hard-fork as a solution to recover the funds, which originated the Ethereum Classic. Figure 3.1 illustrates how reentrancy attacks happen.

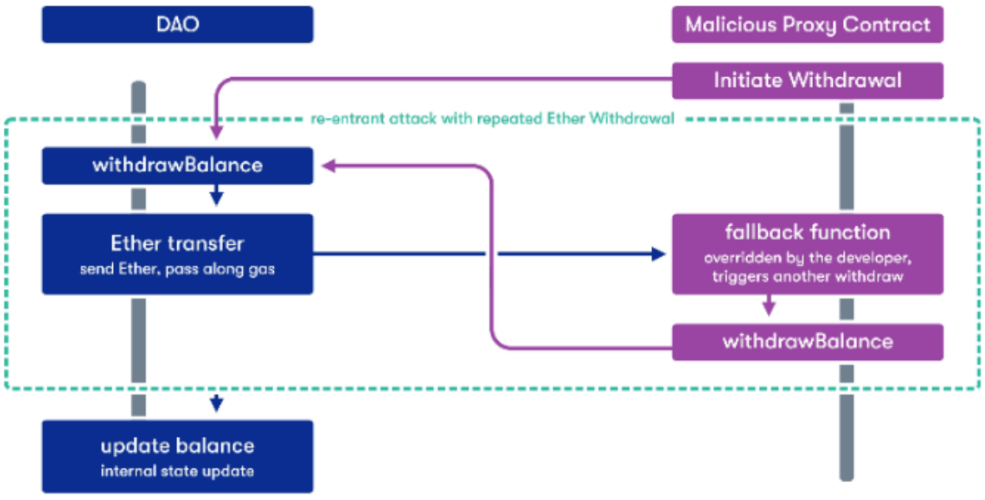

Figure 3.1: Reentrancy attack illustration. **Source:** [22]

3.3.1 What makes this attack possible?

This attack is possible when the state of a contract is not updated before initiating the call to another contract. The other contract can take advantage of this by calling back the parent contract and abusing its outdated state because it is still waiting for the original call to terminate.

[1] https://www.gemini.com/cryptopedia/the-dao-hack-makerdao

3.3.2 How can it be prevented?

Tezos is not affected by the same problem as Ethereum. Internal calls are non-blocking, meaning that they only get executed once the parent operation terminates. As described in section 3.5, Tezos previously suffered from an identical issue, which is now solved.

3.4 Open Proxy Attacks

Open proxy attacks are usually not recurrent. Nevertheless, they can cause critical issues to the application and must get prevented at all costs.

3.4.1 What makes this attack possible?

This attack is usually possible when a given contract receives a callback as a parameter and then uses it to callback another contract. Imagine that some malicious actor forges a callback that points to the target address instead of himself. Consequently, the target address calls itself when resolving the callback, potentially resulting in the loss of funds, as shown in figure 3.2.

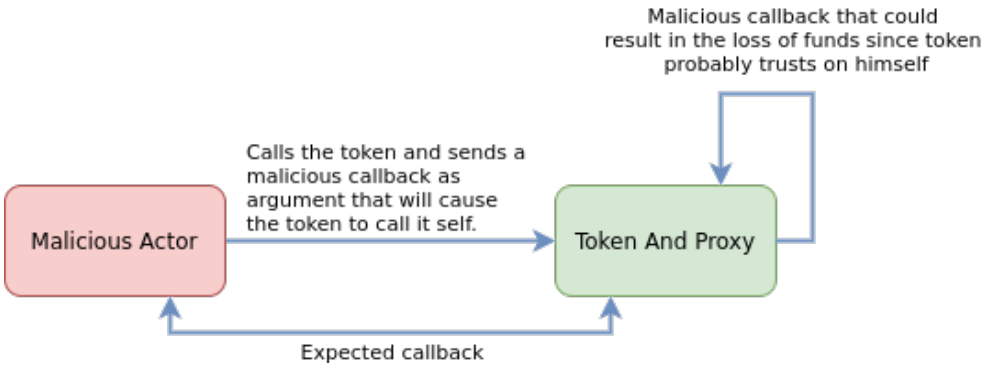

Figure 3.2: Open Proxy problem illustration.

3.4.2 How can it be prevented?

It can be prevented by validating the address associated with the callback, making sure that the sender's address is the same as the address of the callback.

3.5 Execution Order

Execution order used to be a problem in the Tezos blockchain. It initially used **breadth-first search** as shown in table 3.1, which consisted of a "first-in, first-out" approach that led to anti-patterns and originated security flaws similar to the reentrancy bugs. The issue was recently solved when the calling convention was changed to depth-first search as shown in table 3.2, a "last-in, first-out" approach that ensures the correct execution order.

3.5.1 Breadth-first Search

As exemplified in table 3.1, operation **D** gets executed before operation **C** which is older and could cause reentrancy vulnerabilities. A caching mechanism would have to be put in place to protect the contract against reentrancy vulnerabilities.

Executing	Emissions	Resulting Queue	Detail
Initial	A	A	new operation A
A	B, C	B, C	new operations B, C
B, C	D	D, C	new operation D
D, C	-	C	no new operations
C	-	-	no new operations

Table 3.1: Breadth-first search example

3.5.2 Depth-first Search

The depth-first search uses "first-in, last-out" and does not suffer from the problem explained above because the calling convention always resolves the latest operations first.

Executing	Emissions	Resulting Queue	Detail
Initial	A, B	B, A	new operations A, B
B	C, D	D, C, A	new operations C, D
D	-	C, A	no new operations
C	-	A	no new operations
A	-	-	no new operations

Table 3.2: Depth-first search example

3.6 Gas and Storage (Denial of Service)

Gas and Storage limits are also a frequent issue in smart contracts, where the storage of a contract can become so big that any operation will reach the gas limit and fail.

The problem can also happen without any relation to the storage. In this case, the contract logic is so complex that it can reach the gas limit depending on certain conditions and get locked forever without the possibility of recovery.

3.6.1 How can it be prevented?

This kind of issue can be prevented by extensively testing the contract with the insertion of vast amounts of data and seeing how it behaves after each operation, ensuring that most conditions (code branches) get reached and that the growth in size is within the expected threshold.

On **Tezos** it is also possible to use big data structures (called **big maps**), which allow the storage not to get fully deserialized when calling the contract and provide on-demand access to its data.

Chapter 4

Requirements and Design

4.1 Introduction

This chapter describes the requirements and the design of the solution.

- Section [4.2] Presents the requirements of the solution;
- Section [4.3] Describes an high level design of the solution;
- Section [4.4] Contains the main conclusions of this chapter.

4.2 Solution Requirements

This section aggregates the requirements concerning connecting smart contracts to external sources. It divides the requirements per component to facilitate the general understanding of the solution.

4.2.1 Purpose

The general purpose of the solution is to build a secure and reliable system capable of interfacing smart-contracts in the Tezos blockchain with external data sources by leveraging state-of-the-art technology provided by the Chainlink ecosystem.

Figure 4.1 presents the general picture of the intended solution.

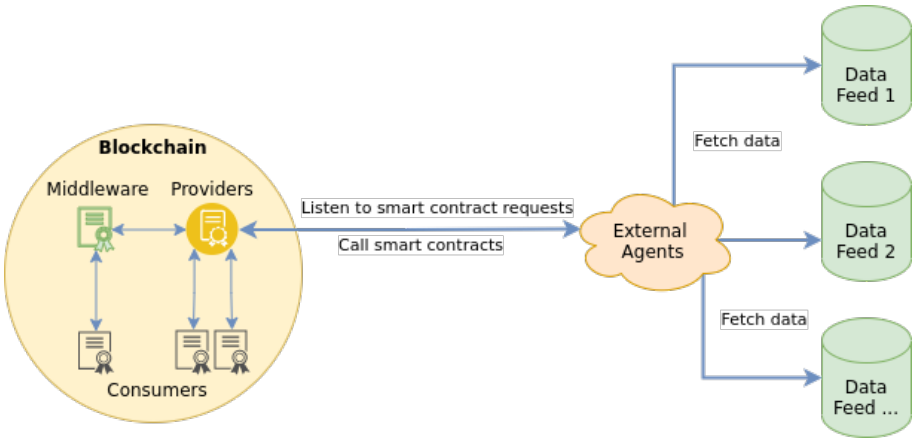

Figure 4.1: A general picture of the solution.

The solution must support the following two models:

- **On-demand request model**: 4.2.2
- **Decentralized data model**: 4.2.3

Including contract administration through the use of a multi-signature contract:

- **Contract administration**: 4.2.4

4.2.2 On-demand request model

This model is the most basic, used for specific use cases where data manipulation is not a considerable risk. Its purpose is to serve as a basic interface between consumer contracts and the outside world.

This model expects four contract kinds, which get divided into two groups, **optional** and **non-optional** contracts.

4.2.2.1 Non-optional contracts:

The model must implement the contracts in figure 4.2, which will be the prominent participants in the model.

1. **Provider** - Oracle contracts, which are used to interface the consumer contracts with the outside world;
2. **Consumer** - Client contracts, which send requests to the providers with the purpose of interacting the external sources.

Figure 4.2: Mandatory contract types in the on-demand request model.

4.2.2.2 Optional contracts:

The model must support two optional contract kinds, used as middleware to give economic incentives and enforce good behavior in the system. By optional, it means that these contracts in figure 4.3 may or may not be part of every use case.

Figure 4.3: Optional contract types in the on-demand request model.

1. **Escrow** - These contracts enforce good behavior between consumers and providers. Their purpose is to hold rewards while requests from consumers do not get answered by providers.
2. **Token** - These are contracts that provide economic incentives to the participants of the system. Providers get rewards from consumers when they answer their requests.

4.2.2.3 Provider requirements:

1. Must be an abstract contract capable of receiving multiple direct or indirect requests;
   - **Indirect:** Requests proxied from white-listed contracts that are not consumers themselves. These can be **Escrow** or **Token** contracts that act as middleware as illustrated in figure 4.4.

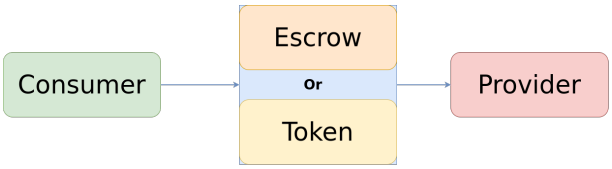

Figure 4.4: Indirect requests in the on-demand request model.

   - **Direct:** Requests coming directly from **Consumer** contracts as illustrated in figure 4.5.
2. After receiving and processing a request, the **Provider** must call back the client with a response;

Figure 4.5: Direct requests in the on-demand request model.

3. When receiving an indirect request, the **Provider** must verify if the request comes from a white-listed address that can either be a **Token** or an **Escrow**. Every indirect request from addresses that are not white-listed must be rejected;

4. If the request comes from a **Token** or **Escrow**, the **Provider** must ensure that the amount included as payment in the request is equal or greater than the payment amount configuration.

5. The contract must support the following administrative actions:

   - Must be possible to disable and enable the acceptance of direct requests;
   - Must be possible to specify the payment amount required for its services;
   - Must be possible to add and remove **Escrow** and **Token** contracts from an white-list;

4.2.2.4 Escrow requirements:

1. Must allow **Consumers** to transmit their requests to a **Provider**.

2. Must freeze **Provider** rewards while the **Consumer** request does not get an answer.

3. Must unfreeze the **Provider** rewards once the response gets transmitted to the **Consumer**.

4.2.2.5 Token requirements:

1. Must implement the **TZIP-12**[1] token standard, a specification for financial application contracts;

2. Must implement the **ERC-677**[2] specification from Ethereum to allow **Consumers** to make payments and transmit their requests to a **Provider** or **Escrow**.

3. The contract must support the following administrative actions:

[1] https://gitlab.com/tezos/tzip/-/blob/master/proposals/tzip-12/tzip-12.md
[2] https://github.com/ethereum/EIPs/issues/677

- Mint tokens;
- Burn tokens;
- Change the administrator address (useful in case the multi-signature contract needs to be updated);
- Pause the contract, useful for maintenance or in case any vulnerability gets detected;

4.2.2.6 Consumer requirements:

1. Must be able to send requests directly to the **Provider**;
2. Must be able to send requests indirectly to the **Provider** or **Escrow** through the **Token** contract by using the **ERC-677** specification;

4.2.3 Decentralized data model

The decentralized model extends the on-demand request model 4.2.2.2 by re-implementing the **Provider** contract as a singleton aggregation contract (an **Aggregator**) to work as a decentralized pool of oracles responsible for feeding prices from external sources into the blockchain environment. The contract aggregates the provisioned prices from multiple white-listed wallets and provides the median result as its latest price.

4.2.3.1 Aggregator requirements:

1. The contract must have a notion of rounds, where each round represents an aggregation attempt;
2. Rounds must have a Time to Live (TTL), enforcing a threshold for price submissions to avoid big price deviations;
3. For a price update to be accepted, a minimum number of submissions must be enforced. The minimum number of submissions is must be a configurable setting;
4. Oracles must be approved before being able to participate in the aggregation process;
5. The contract must provide an entry point to allow oracles to submit data;
6. When oracles submit round prices, they must be rewarded. But only once per round. The reward amount must be a configurable setting;

7. Any oracle shall be able to withdraw their rewards whenever they want;

8. An entry point view must exist to allow client contracts to access the latest price. The view must provide the following information:

    - Latest price;
    - Round Identifier;
    - Timestamp of the latest price update.

9. The administration of the contract in section 4.2.4 must be done through a multi-signature contract (usually called Decentralized Autonomous Organization (DAO), which is composed by multiple entities), the base administrative actions to be supported are:

    - Change the administrator address (useful in case the multi-signature contract needs to be updated);
    - Pause the contract, useful for maintenance or in case any vulnerability gets detected;
    - Changing the minimum number of submissions;
    - Add and remove oracles;
    - Changing the round TTL;

4.2.4 Contract Administration

The solution must provide a multi-signature contract that allows multiple entities to govern/administrate contracts through a voting mechanism. Having a multi-signature contract for ruling contracts provides more trust in the solution since, by design, no single entity will be able to apply changes to any contract without the consensus of other participants.

4.2.4.1 Multi-signature contract requirements:

- Approved entities shall be allowed to propose administrative actions;
- Proposals shall be endorsable by other approved entities;
- Proposals must only be committed once endorsed by the majority of the participants;
- The following administrative action kinds shall be supported:

  - **Internal actions**: Actions that administrate the administrative contract itself;
  - **External actions**: These are actions that administrate other contracts. They must have a generic format to allow any type of administrative operation on target contracts.

- The contract must allow proposals and endorsements to be aggregated off-chain and submitted in a single atomic operation on-chain;

The process shall be the following:

1. A participant creates and signs a proposal to then broadcast it to the remaining participants;
2. The participants receive a proposal receipt, and if they agree with the proposed changes, an endorsement operation is created and signed to then be transmitted back to the proposer;
3. Once the proposer receives enough endorsements, he batches all of them together with a the proposal and commits it on-chain;
4. In a final step, the contract iterates over all endorsements, verifying the signatures against the proposal content, and if every endorsement is valid, including achieving the necessary quorum, the action is committed successfully.

4.3 High level design

This section focuses on describing the system's high-level design. It starts by describing the Multi-signature contract design, a contract used to administrate contracts owned by multiple parties, allowing a decentralized distribution of power.

4.3.1 Multi-signature contract

The multi-signature contract is used to administrate other contracts through a voting mechanism. Its design is generic, allowing arbitrary administrative actions to be performed on a target contract as long as it implements the administration interface.

The contract supports two kinds of actions, internal and external, as shown in figure 4.6. The internal kind is specific for the administration of the contract itself, and the external action is used to administrate the remaining contracts of the system.

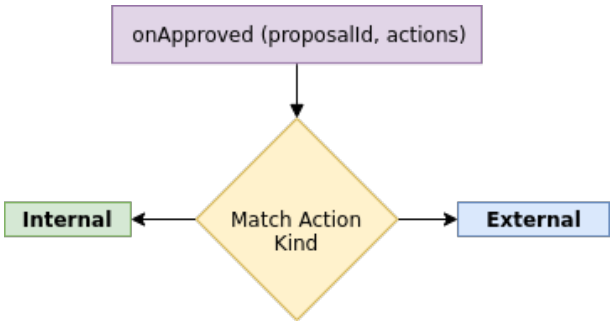

Figure 4.6: Multi-signature actions.

4.3.1.1 Storage interface

The contract storage interface in listing 4.1 shows the storage attributes necessary in the multi-signature contract. Their purpose is described below.

```
# Internal administration action type specification
InternalAdminAction = sp.TVariant(
    changeSigners   = sp.TVariant(
                        removed = sp.TSet(sp.TAddress),
                        added   = sp.TList(
                                    sp.TRecord(
                                        address     = sp.TAddress,
                                        publicKey   = sp.TKey
                                    ).right_comb()
                                )
                    ).right_comb(),
    changeQuorum    = sp.TNat,
    changeMetadata  = sp.TPair(sp.TString, sp.TOption(sp.TBytes)),
).right_comb()

# External administration action type specification
ExternalAdminAction = sp.TRecord(
    target  = sp.TAddress,
    actions = sp.TBytes
).right_comb()

# Proposal action type specification
ProposalAction = sp.TVariant(
    internal = sp.TList(InternalAdminAction),
    external = sp.TList(ExternalAdminAction)
).right_comb()

# Proposal type specification
Proposal = sp.TRecord(
    startedAt       = sp.TTimestamp,
    initiator       = sp.TAddress,
```

```
    endorsements    = sp.TSet(sp.TAddress),
    actions         = ProposalAction
).right_comb()

# Storage type specification
Storage = sp.TRecord(
    quorum          = sp.TNat,
    lastProposalId  = sp.TNat,
    signers         = sp.TMap(
                        sp.TAddress,
                        sp.TRecord(
                            publicKey       = sp.TKey,
                            lastProposalId  = sp.TOption(sp.TNat)
                        ).right_comb()
                    ),
    proposals       = sp.TBigMap(sp.TNat, Proposal),
    activeProposals = sp.TSet(sp.TNat),
    metadata        = sp.TBigMap(sp.TString, sp.TBytes),
).right_comb()
```

Listing 4.1: Multi-signature contract storage interface.

- **quorum** - Is a natural number that specifies the minimum of endorsements each proposal must have before being considered approved;
- **lastProposalId** - A sequential natural number that increments by one when a new proposal is submitted and is used to provide unique integer identifiers to proposals;
- **signers** - A hash map with all approved signers, the key of the map is the signer address, and the value is a record that stores the signer public key and the identifier of the latest submitted proposal;
- **proposals** - A big map that stores proposals submitted by signers. The key is the proposal identifier. The value is a record that stores the proposal action, the endorsements, the signer's address that created the proposal, and the creation date.
- **activeProposals** - A simple set that stores the ongoing proposal identifiers which caches the proposal identifiers to decrease the gas costs;
- **metadata** - It is a big map used for off-chain purposes, where it provides information about the contract to indexers. The key is a string that specifies the field name, and the value is bytes that decode to a UTF-8 string, representing the field value. The value needs to be in bytes format because the blockchain only supports 7-bit ASCII characters in strings.

4.3.1.2 Method interfaces

Since Michelson is statically typed, the multi-signature contract must use bytes to represent external actions. Those bytes are then decoded and processed by the destination, allowing the multi-signature contract to be generic and compatible with any contract that implements the following entry point interface:

```
(bytes %administrate)
```

The **administrate** entry point in figure 4.7 is implemented on the contract where the administrative action will take effect, it expects **bytes** as argument and is called by the multi-sig contract.

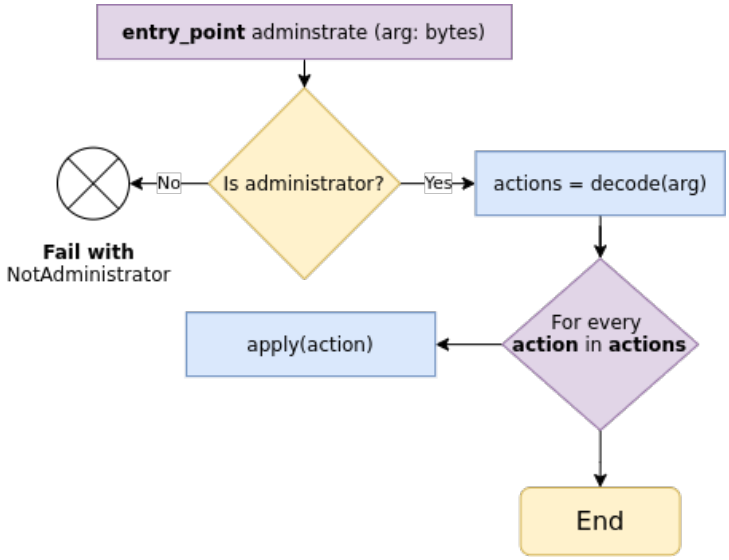

Figure 4.7: Administrate entry point control graph.

The multi-sig specification consists of five primary entry points (functions), specified below:

4.3.1.2.1 "proposal" entry point

An entry point that is used to submit individual proposals of administrative actions. The argument is a variant representing two branches (internal and external), where each branch contains a list of actions that will take effect if the proposal is approved.

```
(or %proposal
    (list %external
        (pair (bytes %actions) (address %target))
    )
```

```
    (list %internal
        (or
            (pair %changeMetadata string (option bytes))
            (or
                (nat %changeQuorum)
                (or %changeSigners
                    (list %added (pair (address %address) (key %
   publicKey)))
                    (set %removed address)
                )
            )
        )
    )
)
```

Listing 4.2: Interface of the "proposal" entry point.

4.3.1.2.2 "endorsement" entry point

An entry point used for endorsing ongoing proposals, where the argument is a list with all the proposal identifiers being endorsed.

```
(list %endorsement nat)
```

Listing 4.3: Interface of the "endorsement" entry point.

4.3.1.2.3 "aggregated_proposal" entry point

Entry point responsible for the submission of proposals in a single batch operation, where the argument is a pair that has as the first element a variant that represents two branches (internal and external) and as the second element the signatures from other participants for endorsing the proposal.

```
(pair %aggregated_proposal
    (or %actions
        (list %external
            (pair (bytes %actions) (address %target))
        )
        (list %internal
            (or
                (pair %changeMetadata string (option bytes))
                (or
                    (nat %changeQuorum)
                    (or %changeSigners
                        (list %added
```

```
                                (pair (address %address) (key %
    publicKey))
                            )
                            (set %removed address)
                        )
                    )
                )
            )
        )
        (pair
            (nat %proposalId)
            (list %signatures
                (pair (signature %signature) (address %signerAddress))
            )
        )
)
```

Listing 4.4: Interface of the "aggregated_proposal" entry point.

4.3.1.2.4 "aggregated_endorsement" entry point

An entry point that allows submitting endorsements in a single batch operation, where the argument is a list of pairs containing the proposal identifier and a list of signatures signed by the participants.

```
(list %aggregated_endorsement
    (pair
        (nat %proposalId)
        (list %signatures
            (pair (signature %signature) (address %signerAddress))
        )
    )
)
```

Listing 4.5: Interface of the "aggregated_endorsement" entry point.

4.3.1.2.5 "cancel_proposal" entrypoint

An entry point used to cancel individual proposals, where the argument is a natural number representing the porposal identifier.

```
(nat %cancel_proposal)
```

Listing 4.6: Interface of the "cancel_proposal" entry point.

4.3.1.3 Preventing Replay Attacks

Aggregation operations include endorsements that are validated and authenticated by using signatures. As discussed previously, the signing mechanism should have measures to prevent repeated signatures, meaning that the content to be signed should always be different.

The multi-sig contract was designed to prevent this vulnerability. The protection consists of adding the proposal identifier, the contract address, and the chain identifier to the signed contents.

Why is it enough, and what do those components mean?

- **Proposal identifier** - Only one endorsement signed by the same key is allowed per proposal. Having the proposal identifier in the signed contents ensures that every new proposal will result in a different signature.
- **Contract address** - Having only the proposal identifier is insufficient when the wallet is used to endorse proposals from other multi-signature contracts. The issue can be solved by adding the contract address to the contents being signed.
- **Chain identifier** - This is already overkill, but having the chain identifier can prevent replay attacks from test chains.

4.3.1.4 Diagrams

The workflows are somewhat extensive, and for that reason, I have included them as appendices.

- Multi-signature proposal: A.1
- Multi-signature endorsement: A.2
- Multi-signature aggregated proposal: A.3
- Multi-signature aggregated endorsement: A.4
- Multi-signature proposal cancellation: A.5

4.3.2 On-Demand Oracle

The system has two different designs, one has an escrow contract that serves as middleware, and the other has not. Both designs have advantages and disadvantages over one another.

Below I explain the designs and also provide some illustrative examples (4.8 and 4.9).

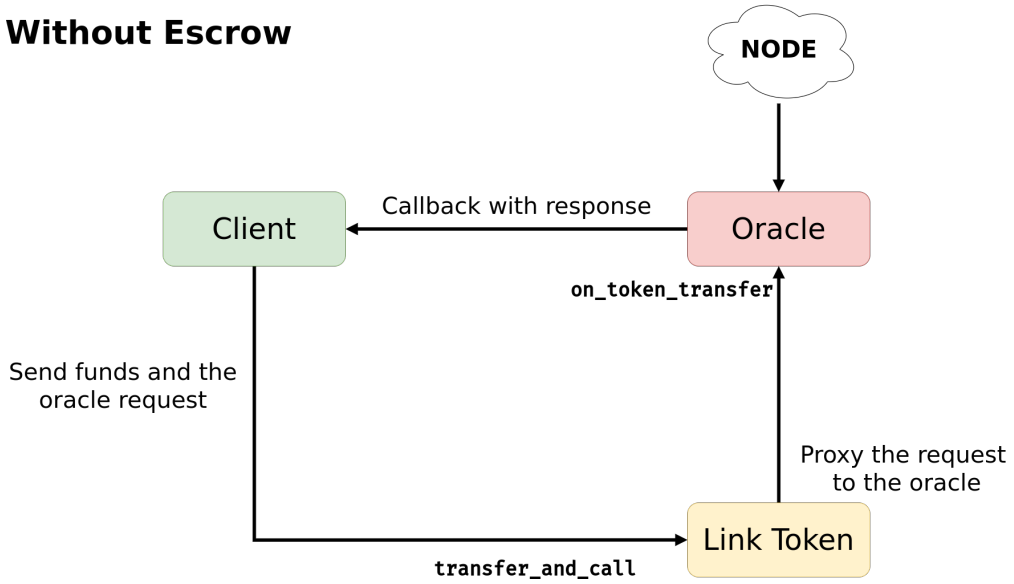

Figure 4.8: On-Demand Oracle without an escrow contract.

The first design comprises three contract abstractions, the oracles, the token, and the clients. It does not contain an escrow service.

In this design, clients send requests to the oracle through the token contract together with the service payment, and oracles send responses directly to the clients.

This design reduces gas costs because there is no escrow. Having an escrow in between would add extra calls that would result in more gas usage.

As a consequence of not having an escrow, the client is forced to believe that the oracle is well-intentioned and that he will complete his part of the agreement by fetching data from outside sources and sending it back to the client.

The escrow increases clients' trust when sending requests to the oracles because the escrow locks oracle payments and unlocks them only when they send a response. Thus, if the oracle doesn't send a reply and the request times out, the client is free to cancel his request and refund the payment.

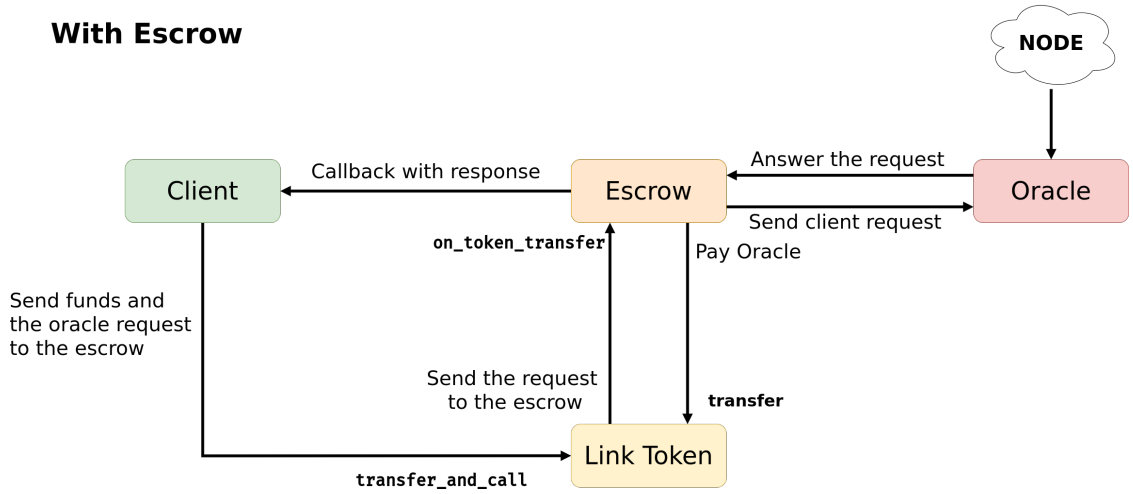

Figure 4.9: On-Demand Oracle with an escrow contract.

4.3.2.1 Token contract

4.3.2.1.1 Storage interface

The contract storage interface in listing 4.7 shows the storage attributes necessary in the token contract. Their purpose is described below.

```
sp.TRecord(
    administrator = sp.TAddress,
    all_tokens = sp.TNat,
    ledger = sp.TBigMap(
        sp.TPair(sp.TAddress, sp.TNat),
        sp.TRecord(
            balance = sp.TNat
        ).layout("balance")
    ),
    metadata = sp.TBigMap(sp.TString, sp.TBytes),
    operators = sp.TBigMap(
        sp.TRecord(
            operator = sp.TAddress,
            owner = sp.TAddress,
            token_id = sp.TNat
        ).layout(("owner", ("operator", "token_id"))),
        sp.TUnit
    ),
    paused = sp.TBool,
    token_metadata = sp.TBigMap(
        sp.TNat,
        sp.TRecord(
            token_id = sp.TNat,
            token_info = sp.TMap(sp.TString, sp.TBytes)
```

```
        ).layout(("token_id", "token_info"))
    )
).layout((("administrator", ("all_tokens", "ledger")), (("metadata"
    , "operators"), ("paused", "token_metadata"))))
```

Listing 4.7: Token contract storage interface

- **administrator** - This attribute contains the address of the contract administrator;
- **all_tokens** - A natural number that provides the number of existing distinct tokens minted by the contract;
- **ledger** - The ledger attribute is a big map that stores the ownership of tokens, where the key is a pair composed by the owner address and the token identifier, and its value is the balance owned;
- **metadata** - It is a big map used for off-chain purposes, where it provides information about the contract to indexers. The key is a string that specifies the field name, and the value is bytes that decode to a UTF-8 string, representing the field value. The value needs to be in bytes format because the blockchain only supports 7-bit ASCII characters in strings.
- **operators** - The operators attribute stores permissions used to permit addresses to operate the balance of other addresses;
- **paused** - The attribute is used for enabling or disabling the contract. When disabled, only administration operations are allowed;
- **token_metadata** - Similar to the attribute **metadata**, **token_metadata** is also used for off-chain purposes, where it provides information specific to a given token (e.g., the number of decimals places, name, description, and icon).

4.3.2.1.2 Method interfaces

Most methods are inherited from **TZIP-12**[3] specification. The implementation will add a new entry point named **transfer_and_call**, which the clients will use to transfer the ownership of tokens from one address to another and transmit information to the recipient when the transfer is successful. The argument is a list of transactions containing the amount to be transferred, the recipient address, the token identifier, and the payload to be sent to the recipient.

[3]https://gitlab.com/tezos/tzip/-/blob/master/proposals/tzip-12/tzip-12.md#interface-specification

```
(list %transfer_and_call
  (pair
    (address %from_)
    (list %txs
      (pair
        (address %to_)
        (pair
          (address %callback)
          (pair
            (bytes %data)
            (pair
              (nat %token_id)
              (nat %amount)
)))))))
```

Listing 4.8: Interface of the "transfer_and_call" entry point in the token contract.

4.3.2.2 Oracle contract

4.3.2.2.1 Storage interface

The contract storage interface in listing 4.9 shows the storage attributes necessary in the oracle contract. Their purpose is described below.

```
# Parameter type specification
Parameter = sp.TVariant(
    int     = sp.TInt,
    string  = sp.TString,
    bytes   = sp.TBytes
).right_comb()

# Parameters type specification
Parameters              = sp.TMap(sp.TString, Parameter)

# Request type specification
Request = sp.TRecord(
    request = sp.TRecord(
        id                  = sp.TNat,
        jobId               = sp.TString,
        clientAddress       = sp.TAddress,
        callbackAddress     = sp.TAddress,
        cancelTimeout       = sp.TTimestamp,
        parameters          = Parameters,
    ).right_comb(),
    payment = sp.TNat
```

```
).right_comb()

# Storage type specification
sp.TRecord(
    config = sp.TRecord(
        adminAddress        = sp.TAddress,
        tokenAddress        = sp.TAddress,
        active              = sp.TBool,
        minCancelTimeout    = sp.TNat,
        minPayment          = sp.TNat,
    ),
    requests = sp.TBigMap(sp.TBytes, Request),
    metadata = sp.TBigMap(sp.TString, sp.TBytes),
).right_comb()
```

Listing 4.9: Oracle contract storage interface

- **config** - Stores the contract configurations.

  Attributes:

  - **adminAddress** - The administrator address;
  - **tokenAddress** - The address of the token contract;
  - **active** - Disables or enables the orable;
  - **minCancelTimeout** - Specifies the minimum acceptable **TTL**;
  - **minPayment** - Specifies the minimum acceptable payment amount;

- **requests** - A big map that stores requests from clients, where the key is a **blake2b** hash of the value, and the value is composed of the request data and the **payment** amount sent by the client. Off-chain agents will listen to this attribute to extract requests sent from clients and then apply actions off-chain.

  Request attributes:

  - **id** - A natural number identifier sent by the client;
  - **jobId** - A hexadecimal identifier of a Chainlink job;
  - **clientAddress** - The original client address;
  - **callbackAddress** - A callback address used to send the response to the client or escrow;
  - **cancelTimeout** - The TTL of the request;
  - **parameters** - Request query, it can be a string, an integer or bytes;

- **metadata** - It is a big map used for off-chain purposes, where it provides information about the contract to indexers. The key is a string that specifies the field name, and the value is bytes that decode to a UTF-8 string, representing the field value. The value needs to be in bytes format because the blockchain only supports 7-bit ASCII characters in strings.

4.3.2.2.2 Method interfaces

Entry point **on_token_transfer** is used when the client does not want to use an escrow contract and instead sends the request through the token.

```
(pair %on_token_transfer
    (nat %amount)
    (pair
        (bytes %data)
        (pair
            (address %sender)
            (nat %tokenId)
)))
```

Listing 4.10: Interface of the "on_token_transfer" entry point in the oracle contract.

Entry point **create_request** is used by the escrow contract to transmit requests from clients.

```
(pair %create_request
    (nat %payment)
    (pair %request
        (address %callbackAddress)
        (pair
            (timestamp %cancelTimeout)
            (pair
                (address %clientAddress)
                (pair
                    (nat %id)
                    (pair
                        (string %jobId)
                            (pair
                                (address %oracleAddress)
                                (map %parameters
                                    string
                                    (or (bytes %bytes) (or (int %
   int) (string %string)))
))))))))
```

Listing 4.11: Interface of the "create_request" entry point in the oracle contract.

4.3.2.3 Escrow contract

4.3.2.3.1 Storage interface

```
# LockedRequest type specification
LockedRequest = sp.TRecord(
    payment         = sp.TNat,
    oracleAddress   = sp.TAddress,
    clientAddress   = sp.TAddress,
    callbackAddress = sp.TAddress,
    cancelTimeout   = sp.TTimestamp
)

# Storage type specification
sp.TRecord(
    adminAddress  = sp.TAddress,
    tokenAddress  = sp.TAddress,
    locked        = sp.TBigMap(sp.TPair(sp.TAddress, sp.TNat),
   LockedRequest)
    metadata      = sp.TBigMap(sp.TString, sp.TBytes)
).right_comb()
```

Listing 4.12: Escrow contract storage interface

- **adminAddress** - The administrator address;
- **tokenAddress** - The token address;
- **locked** - Is a big map that stores the requests transmitted from the client to the oracle contract while those do not get fulfilled. The key is composed by the client address and the request identifier–the value is part of the request data sent by the client.

  Request attributes:

  - **payment** - The amount to be sent to the oracle once he fulfills the request;
  - **clientAddress** - The oracle address;
  - **clientAddress** - The original client address;
  - **callbackAddress** - A callback address used to send the response to the client;
  - **cancelTimeout** - The TTL of the request;
- **metadata** - It is a big map used for off-chain purposes, where it provides information about the contract to indexers. The key is a string that specifies the

field name, and the value is bytes that decode to a UTF-8 string, representing the field value. The value needs to be in bytes format because the blockchain only supports 7-bit ASCII characters in strings.

4.3.2.3.2 Method interfaces

Entry point **fulfill_request** is used by oracles to fulfill client requests and claim the payment.

```
(pair %fulfill_request
    (address %clientAddress)
    (pair
        (map %parameters
            string
            (or (bytes %bytes) (or (int %int) (string %string)))
        )
        (nat %requestId)
))
```

Listing 4.13: Interface of the "fulfill_request" entry point in the escrow contract.

Entry point **on_token_transfer** is similar to the oracle, but in this case, the payment gets locked on the escrow side and is only sent to the oracle once he claims the reward.

```
(pair %on_token_transfer
    (nat %amount)
    (pair
        (bytes %data)
        (pair
            (address %sender)
            (nat %tokenId)
)))))
```

Listing 4.14: Interface of the "on_token_transfer" entry point in the escrow contract.

4.3.3 Decentralized model

The decentralized model is more secure than the on-demand oracle model explained above because it comprises multiple entities that submit data to the contract, where the data gets aggregated and computed. The information gets aggregated in rounds. Clients can always request the information computed in the latest round by paying a given amount to the contract that gets distributed between the entities participating in the aggregation process. Figure 4.10 illustrates the scenario.

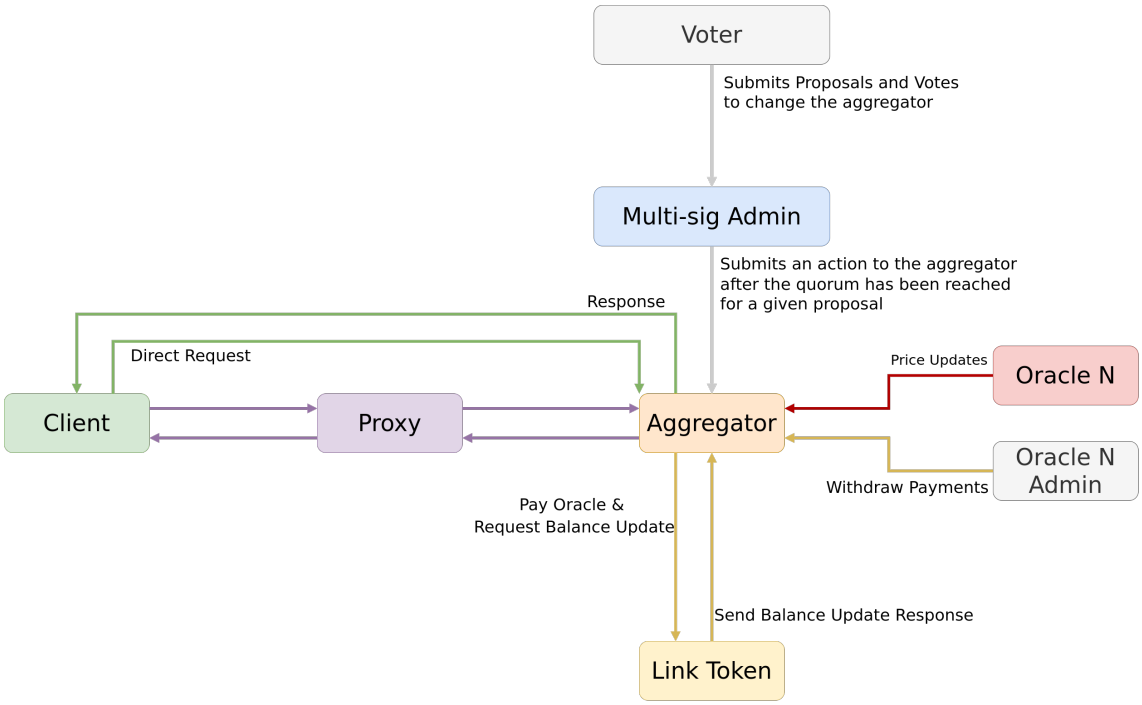

Figure 4.10: Decentralized model diagram.

4.3.3.1 Aggregator contract

The main component of the decentralized price feed is the aggregation contract, which relies on multiple agents that gather data about the current prices of precious metals, fiat, or other cryptocurrencies and submit them into the contract as shown in appendix B.1. The contract then stores the submissions in a map and recalculates the median price of the round submissions.

A new round can start at any time when one of the following conditions occur:

- **Time based** - A new round can start every **X** minutes, but the countdown resets every time a new round starts;
- **Price based** - When price deviates from a given percentage when compared to the latest price;

4.3.3.1.1 How is the median obtained?

If the number of submissions is odd, then the median will be the middle of a sorted list of all submissions. But if the number of submissions is even, then the average of the two middle indexed items becomes the median price.

4.3.3.1.2 Client Requests

Clients can obtain the latest price by calling the aggregator directly as shown in 4.11 or by calling a proxy contract that points to the aggregator as shown in 4.12.

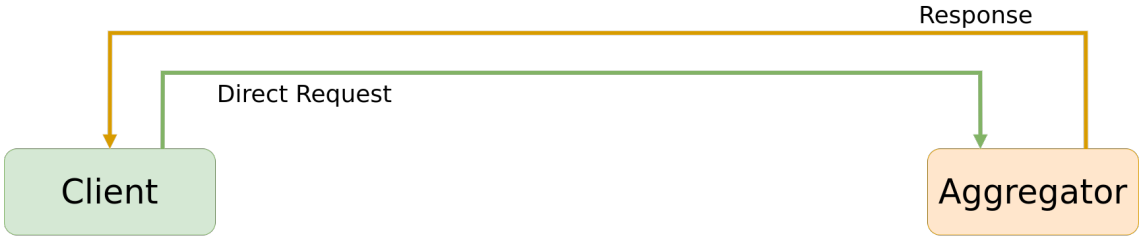

Figure 4.11: Client request (Direct)

Figure 4.12: Client request (Proxied)

4.3.3.1.3 Storage interface

The contract storage interface in listing 4.15 shows the storage attributes necessary in the aggregator contract. Their purpose is described below.

```
# Round type specification
Round = sp.TRecord(
    roundId          = sp.TNat,
    answer           = sp.TNat,
    startedAt        = sp.TTimestamp,
    updatedAt        = sp.TTimestamp,
    answeredInRound  = sp.TNat
).right_comb()

# Round details type specification
RoundDetails = sp.TRecord(
    submissions      = sp.TMap(sp.TAddress, sp.TNat),
    minSubmissions   = sp.TNat,
    maxSubmissions   = sp.TNat,
    timeout          = sp.TNat,
    activeOracles    = sp.TSet(sp.TAddress)
).right_comb()

# Oracle details type specification
OracleDetails = sp.TRecord(
    startingRound        = sp.TNat,
    endingRound          = sp.TNat,
```

```
    lastStartedRound    = sp.TNat,
    lastReportedRound   = sp.TNat,
    withdrawable        = sp.TNat,
    adminAddress        = sp.TAddress
).right_comb()

# Storage type specification
sp.TRecord(
    active                = sp.TBool,
    decimals              = sp.TNat,
    adminAddress          = sp.TAddress,
    tokenAddress          = sp.TAddress,
    metadata              = sp.TBigMap(sp.TString, sp.TBytes),
    minSubmissions        = sp.TNat,
    maxSubmissions        = sp.TNat,
    restartDelay          = sp.TNat,
    timeout               = sp.TNat, # In minutes
    oraclePayment         = sp.TNat,
    latestRoundId         = sp.TNat,
    reportingRoundId      = sp.TNat,
    rounds                = sp.TBigMap(sp.TNat, Round),
    previousRoundDetails  = RoundDetails,
    reportingRoundDetails = RoundDetails,
    recordedFunds         = sp.TRecord(
                                available   = sp.TNat,
                                allocated   = sp.TNat
                            ),
    oracles               = sp.TMap(sp.TAddress, OracleDetails)
).right_comb()
```

Listing 4.15: Aggregator contract storage interface

- **active** - Disabled or enables the contract;
- **decimals** - The number of decimals of the answer. Necessary because there is no floating point numbers in smart contracts;
- **adminAddress** - The address of the administrator contract;
- **tokenAddress** - The address of the token contract;
- **metadata** - It is a big map used for off-chain purposes, where it provides information about the contract to indexers. The key is a string that specifies the field name, and the value is bytes that decode to a UTF-8 string, representing the field value. The value needs to be in bytes format because the blockchain only supports 7-bit ASCII characters in strings.
- **minSubmissions** - The minimum amount of submissions required per round;

- **maxSubmissions** - The maximum amount of submissions per round;
- **restartDelay** - Defines a threshold enforcing that a round cannot start within a given time interval from the previous round.
- **timeout** - Defines the round timeout. If a round is not completed after a given period, the current round gets terminated and a new round starts;
- **oraclePayment** - Specifies the amount paid to oracles when they submit data. Occurs once per round.
- **latestRoundId** - Stores the round identifier of the latest round to be completed;
- **reportingRoundId** - Stores the identifier of the current round;
- **rounds** - A big map containing historical data of each round ;

  Round attributes:

  - **roundId** - Round identifier;
  - **answer** - Contains the aggregation result. It is only fulfilled when the number of submissions is equal or bigger than **minSubmissions**.
  - **startedAt** - Timestamp of the round started;
  - **updatedAt** - Timestamp of when the latest answer was submitted;
  - **answeredInRound** - The identifier of the round where the latest answer was submitted;

- **previousRoundDetails** - Previous round details;
- **reportingRoundDetails** - Current round details;

  Round detail attributes:

  The round details contain cloned fields from the root record–they are necessary because rounds must not be updated once the round starts, and the values in the root record can be updated at any time.

  - **submissions** - Oracle submissions in the round;
  - **minSubmissions** - A clone of the **minSubmissions** field when the round started;
  - **maxSubmissions** - A clone of the **maxSubmissions** field when the round started;
  - **timeout** - A clone of the **timeout** field when the round started;

  - **activeOracles** - A clone of the **oracles** field when the round started.

- **recordedFunds** - Stores the number of funds owned by the contract, where allocated funds are payments not yet claimed by oracles;
- **oracles** - A map containing all oracles participating in the aggregation. Oracles are indexed by their addresses.

  Oracle attributes:

  - **startingRound** - The round identifier when the oracle added;
  - **endingRound** - The round identifier when the oracle will disabled;
  - **lastStartedRound** - The latest round started by the baker;
  - **lastReportedRound** - The latest round where the baker submitted an answer;
  - **timeout** - The amount of uncollected payments;
  - **adminAddress** - The address of the administrator, which can be the contract address.

Method interfaces

Entry point **submit** is used by external adapters to submit answers to the current round.

```
(pair %submit nat nat) # Round identifier on the left and answer on
    the right
```

Listing 4.16: Interface of the "submit" entry point in the aggregator contract.

Entry point **latest_round_data** is used by clients to obtain the latest round data.

```
(address %latest_round_data)
```

Listing 4.17: Interface of the "latest_round_data" entry point in the aggregator contract.

4.3.3.1.4 Diagrams

The diagrams are somewhat extensive, and for that reason, I have included them as appendices.

- Price Submission: B.1
- Proxied Request: B.2

4.4 Conclusion

I believe that the requirements and design of the solution were adequately defined, starting with the aggregation of requirements and then the definition of a high-level specification. However, it is natural to think that a few things may still change in the future as the solution evolves.

A friend once told me the following:

> *"Every non-trivial project needs to reinvent itself at least once."*

Chapter 5

Implementation Overview

5.1 Introduction

This chapter presents a brief overview of the implementation process by explaining how it was structured and the technologies used.

- Section [5.2] - Implementation structure;
- Section [5.4] - Technologies used;

5.2 Implementation Structure

The project uses GitLab as a version control tool, where all the code lives in a single repository (mono-repository). It leverages **Make** to compose multiple targets used to install, compile, test, and deploy parts of the solution independently.

5.2.1 Tasks

The implementation was structured in the following way:

1. Pipeline preparation, which consisted in the creation of a Makefile with targets to prepare the environment, compile and test the contracts;
2. Implementation of the contracts for the on-demand request model;
3. Implementation of the contracts for the decentralized model;
4. Chainlink node setup;

5. Implementation of a driver for listening to smart contracts and call the Chainlink node;
6. Implementation of an oracle agent used to connect the Chainlink node with smart contracts;
7. Creation of job specs used by the Chainlink node;
8. Preparation of a sandbox to emulate the production environment. Described in 6.2;
9. Implementation of a command-line interface to communicate with the smart-contracts;
10. Creation of a small frontend used as proof of concept for the solution;
11. Document the solution.

5.3 Tools Implementation

5.3.1 External initiator driver

The external initiator is written in Golang. It listens to specific contract operations by looking into the operation metadata and checking if an operation destination is a monitored contract and if the operation targets the "request" or "on_token_transfer" entry-points.

If the operation matches the condition, the initiator parses the storage difference to extract the request parameters and then calls the Chainlink node to dispatch an action to the external agent.

Figure 5.1 shows the external initiator interactions with other components.

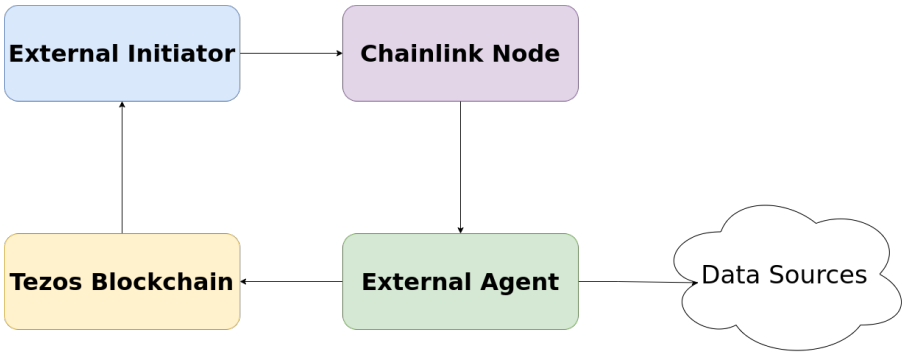

Figure 5.1: External initiator interactions.

5.3.2 External agent

The external agent is implemented in Typescript and consists of a simple web server with a single end-point /**submit** that the Chainlink node can call to request the submission of data into the smart contract as shown if figure 5.1. It can fetch data from three data sources **Kraken**, **Coingecko**, and **Coinbase**, which are configured by the user.

Environment variables:

```
# Web server port (Optional: 3000 is the default)
PORT=<...>
# Price API (Optional: COINBASE is the default)
PRICE_SOURCE=<KRAKEN|COINGECKO|COINBASE>
# Private key
SECRET_KEY=edsk<...>
# Tezos RPC
TEZOS_RPC=https://<...>
```

Listing 5.1: Environment variables.

Web server end-points:

```
{
    id: string;
    data: {
        aggregatorAddress: string;
    },
};
```

Listing 5.2: POST body of the "/submit" end-point.

Web server responses:

```
{
    jobRunID: string; // Job ID that dispatched the request
    result: string; // The hash of the onchain operation.
    statusCode: 200;
}
```

Listing 5.3: Success response.

```
{
    jobRunID: string; // Job ID that dispatched the request
    status: 'errored';
```

```
    error: {
        name: string;
        message: string;
        errors: string[]; // Error details
    },
    statusCode: 500;
}
```

Listing 5.4: Success response.

5.3.3 Command-line interface

The command-line interface is written in Typescript. It allows the user to call smart contracts from the terminal and is heavily used by the continuous integration pipeline.

```
# Deploy contracts related to the decentralized model
./cli.js originate-contracts --config "config/pricefeed.yaml"
```

Listing 5.5: CLI usage example.

It is composed of three components:

1. **Call builder** - Constructs the entry point calls that are sent to smart contracts;
2. **Commander** - Reads the user inputs and performs the requested actions;
3. **Packer** - Encodes **Michelson** instructions into bytes. Necessary in the multi-signature contract, where external actions need to be generic by being encoded as bytes.

5.4 Technologies Used

- **Chainlink** - It offers a set of tools for node operators to connect smart contracts to external data sources. The tools are mainly compatible with blockchains built on top of Ethereum Virtual Machine (EVM) but are modular enough to support the Tezos blockchain. I took responsibility for adding a generic external initiator driver that listens to Tezos transactions, parses their content, and dispatches actions based on the configured job specification;

- **SmartPy and Ocaml** - SmartPy is a high-level programming language for writing and testing smart contracts on Tezos efficiently. The compiler is written in Ocaml, which makes Ocaml also a direct dependency;
- **Typescript** - It is being used to write the external agent and the command-line interface that injects operations into the blockchain and the front-end tools for demo purposes. Typescript allows writing cleaner and easier to use code. Writing in pure javascript can be cumbersome since it lacks type checking, and Typescript is a good solution since it is a superset of javascript that provides types on top;
- **Golang** - Mainly used to add extra functionalities to the chainlink toolkit, like the external initiator mentioned above;
- **React** - Web framework used to write the UI/UX for interacting with the smart contracts;
- **Docker and Make** - Tools used for the development and testing environments, multiple containers will encapsulate the necessary tools.

5.5 Conclusion

The implementation phase had a few challenges related to the integration between multiple contracts and the listening of contract transactions from off-chain agents, which got surpassed. Figure 5.2 shows a demo of the solution–it presents a decentralized model composed of 6 oracles provisioning price data into the aggregator contract every five minutes.

Figure 5.2: Solution demo.

Chapter 6

Testing

6.1 Introduction

This section details the testing procedures taken to ensure and validate the quality of the solution.

The testing followed a Test Driven Development (TDD) approach, where most tests got implemented before the features. It enforced a better coverage of the solution and provided a more reliable development experience.

The solution has a continuous integration pipeline, which runs against every pull request and is composed of three jobs (**contracts/tools compilation**, **contracts testing**, and **end-to-end testing**).

Figure 6.1 shows the pipeline during a run of a merge request.

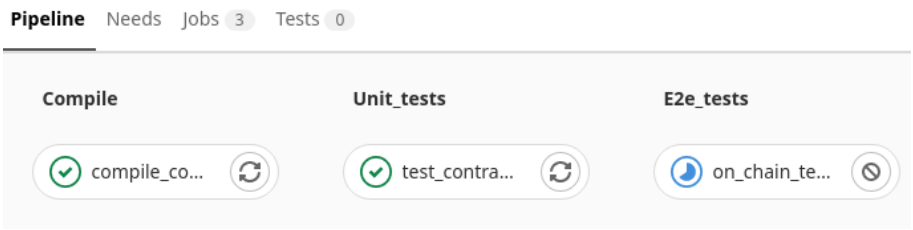

Figure 6.1: Continuous integration pipeline.

6.1.1 Types of testing in use

- **Unit tests** - Test the logic of each contract entry point;
- **Functional tests** - Test the integration between multiple contracts against the specification;

- **Regression tests** - Verify differences between committed and non-committed compilation and test outputs;
- **End-to-end tests** - Test the integration between multiple contracts and tools in a sandbox environment.

6.1.2 Tasks Structure

- Section [6.2] Sandbox setup;
- Section [6.3] Smart contract testing;
- Section [6.4] End-to-end testing;

6.1.3 Objectives

- Preparation of a sandbox to emulate the production environment. It provides trustable test baselines and enables the tests to run in a continuous integration pipeline on every pull request;
- Test implementation for smart contracts and tools, ensuring the solution gets adequately tested;
- Prepare a continuous integration pipeline to run the whole battery of tests on every pull request and in the **main** branch;
- Ensure good coverage of the whole solution.

6.2 Sandbox setup

This section describes the setup of a sandbox that emulates the production environment.

Having a sandbox similar to the production environment provides high-quality test baselines, facilitates the development, and allows the tests to run in a continuous integration pipeline on every pull request;

6.2.1 Sandbox specification:

The sandbox is composed of several docker containers, which connect to a single network. Figure 6.2 illustrates the communication workflow between components,

and below is a description of each component and its role in the system.

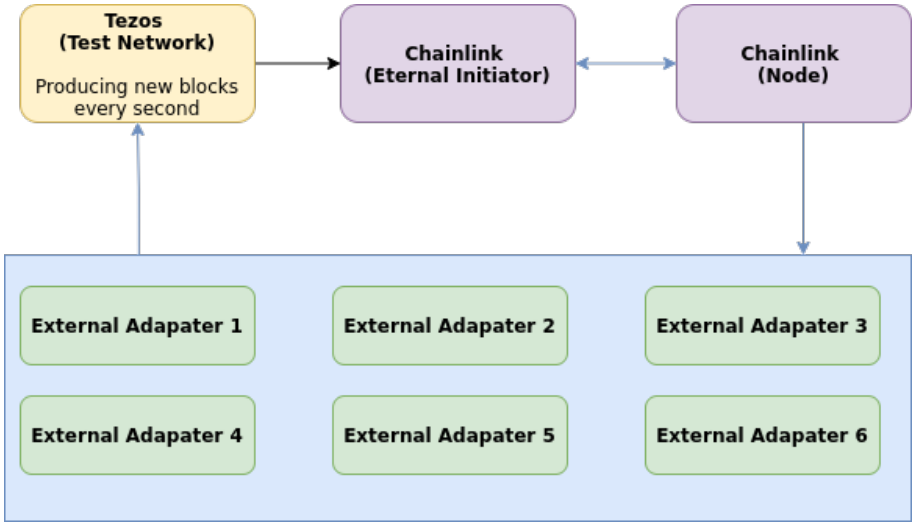

Figure 6.2: Sandbox components.

Components:

- **Tezos (Test Network)** - This component deploys a Tezos test network running the latest protocol, composed of two validators, and produces a new block every minute. All smart contracts get deployed in this network.
- **Chainlink (External Initiator)** - It is a Chainlink tool that runs a **Tezos** driver explicitly developed for this solution, which listens to the contracts and dispatches actions that get transmitted to the **external adapter**.
- **External Adapters** - This tool was developed specifically for this project, and its purpose is to communicate with external APIs and call smart contracts on the Tezos blockchain. In figure 6.2 each external adapter represents an oracle agent that fetches information off-chain and submits it to the contract.

6.3 Contract testing

The unit and functional testing of contracts are composed of 2 steps performed against all implemented contracts. It first checks the compilation of contracts and then runs test various scenarios.

- **Compilation to Michelson** - This step ensures that there are not compilation errors, meaning that the high-level code is valid.

```
import smartpy as sp

Client = sp.io.import_script_from_url("file:contracts/
    price_feed/client.py").Client

####################
# Compilation Targets
####################

sp.add_compilation_target(
    "price_feed_client",
    Client(
        admin = sp.address("KT1_ADMIN_ADDRESS"),
        proxy = sp.address("KT1_PROXY_ADDRESS"),
    )
)
```

Listing 6.1: Compilation example

- **Test of scenarios** - This step runs test scenarios on an internal interpreter, where various conditions get asserted, including the interaction with other contracts. This step does not run on the blockchain.

```
import smartpy as sp

Client = sp.io.import_script_from_url("file:contracts/
    price_feed/client.py").Client

@sp.add_test(name = "Test Client")
def testClient():
    scenario = sp.test_scenario()

    scenario.h2("Originate client")
    client = Client(
        admin = sp.address("KT1_ADMIN_ADDRESS"),
        proxy = sp.address("KT1_PROXY_ADDRESS"),
    )
    scenario += client

    scenario.h2("Administrate client")
    client.administrate(
        admin = sp.address("KT1_ADMIN_ADDRESS_NEW"),
        proxy = sp.address("KT1_PROXY_ADDRESS_NEW"),
    ).run(sender = sp.address("KT1_ADMIN_ADDRESS")
```

Listing 6.2: Test scenario example

6.4 End-to-end testing

The end-to-end testing uses the sandbox setup explained above in section 6.2 to deploy a test network and various other tools that interface the smart contracts with external APIs.

1. It starts by bootstrapping the sandbox containers;
2. Setups the Chainlink node and initiator;
3. Compiles the contracts;
4. Deploys the contracts in a test blockchain;
5. Does functional testing to the solution by calling each contract individually and checking that the expected action was performed;
6. Validates the final state of the contracts.

6.5 Conclusion

Test automation is fundamental to a project like this. It brings many benefits, including keeping a certain level of quality by forcing the implementation to comply with given requirements, avoiding regressions, and excluding trust on a human level. It also facilitates the developer's work by reducing the amount of manual testing and providing helpful feedback while implementing, maintaining, and reviewing features.

Chapter 7

Conclusions and Future Work

This last chapter assembles the main conclusions taken from the internship and aggregates objectives for future work.

7.1 Conclusion

It became clear that blockchains are a promising technology much-discussed nowadays and can initiate a new era of data storage and code execution by leveraging smart contracts, changing how people do business. Layer two technologies like off-chain data provision and scaling solutions are pivotal for the future of blockchain applications and an excellent area for research.

The technology is still early in adoption and has seen tremendous technological advancements in the last few years. Nevertheless, it can still fail due to the possibility of new regulations or people's skepticism in adopting the technology. Furthermore, central banks are starting to experiment with their own Central Bank Digital Currency (CBDC), where China already has a pilot project in use by a percentage of their public workers. So, this leads me to believe that once the currency part of the technology becomes standard, smart contracts will be the norm for service payments.

7.2 Future work

For future work, the plan is to keep evolving the **SmartPy** platform by improving the current features, user experience, and adapting new features as they get included in Tezos.

Additionally, I plan to finalize the bridge between Ethereum and Tezos and integrate on-chain views into the existing contracts once the new protocol **Hangzhou** gets adopted.

References

[1] Victor Allombert, Mathias Bourgoin, and Julien Tesson. Introduction to the tezos blockchain, 2019.

[2] Jacob Arluck. Liquid proof-of-stake. https://medium.com/tezos/liquid-proof-of-stake-aec2f7ef1da7, 2020.

[3] Lăcrămioara Aştefanoaei, Pierre Chambart, Antonella Del Pozzo, Thibault Rieutord, Sara Tucci, and Eugen Zălinescu. Tenderbake – a solution to dynamic repeated consensus for blockchains, 2021.

[4] Abdeljalil Beniiche. A study of blockchain oracles. https://www.researchgate.net/publication/340662783_A_Study_of_Blockchain_Oracles, 2020.

[5] Bruno Bernardo, Raphaël Cauderlier, Guillaume Claret, Arvid Jakobsson, Basile Pesin, and Julien Tesson. Making tezos smart contracts more reliable with coq. *Leveraging Applications of Formal Methods, Verification and Validation: Applications*, page 60–72, 2020.

[6] Bruno Bernardo, Raphaël Cauderlier, Zhenlei Hu, Basile Pesin, and Julien Tesson. Mi-cho-coq, a framework for certifying tezos smart contracts, 2019.

[7] Chainlink. Asset tokenization: Bringing real-world value to the blockchain. https://blog.chain.link/asset-tokenization-bringing-real-world-value-to-the-blockchain.

[8] Deloitte. The tokenization of assets is disrupting the financial industry. are you ready? https://www2.deloitte.com/content/dam/Deloitte/lu/Documents/financial-services/lu-tokenization-of-assets-disrupting-financial-industry.pdf, 2020.

[9] Nate DiCamillo. Vertalo, tzero are bringing $300m in real estate to the tezos blockchain. https://www.coindesk.com/vertalo-tzero-are-bringing-300m-in-real-estate-to-the-tezos-blockchain, 2020.

[10] Stefan Dziembowski, Sebastian Faust, and Kristina Hostáková. General state channel networks. https://eprint.iacr.org/2018/320.pdf, 2018.

[11] Alexander Egberts. The oracle problem - an analysis of how blockchain oracles undermine the advantages of decentralized ledger systems. *Organizations & Markets: Policies & Processes eJournal*, 2017.

[12] Noama Fatima Samreen and Manar H. Alalfi. Reentrancy vulnerability identification in ethereum smart contracts. *2020 IEEE International Workshop on Blockchain Oriented Software Engineering (IWBOSE)*, Feb 2020.

[13] Tim Fries. Securitize and elevated returns to tokenize $1 billion of real estate assets on tezos. https://tokenist.com/securitize-and-elevated-returns-to-tokenize-1-billion-of-real-estate-assets-on-te 2021.

[14] LM Goodman. Tezos — a self-amending crypto-ledger white paper. https://tezos.com/whitepaper.pdf, 2014.

[15] BitFury Group. Proof of stake versus proof of work. https://bitfury.com/content/downloads/pos-vs-pow-1.0.2.pdf, 2015.

[16] Bitfury Group. Land-titling project will extend to other government departments and increase blockchain capabilities for georgian citizens, 2016.

[17] Lewis Gudgeon, Pedro Moreno-Sanchez, Stefanie Roos, Patrick McCorry, and Arthur Gervais. Sok: Layer-two blockchain protocols. In Joseph Bonneau and Nadia Heninger, editors, *Financial Cryptography and Data Security*, pages 201–226, Cham, 2020. Springer International Publishing.

[18] Weilian Xue Honglei Li. A blockchain-based sealed-bid e-auction scheme with smart contract and zero-knowledge proof. Security and Communication Networks, vol. 2021, Article ID 5523394, 10 pages, 2021, 2021.

[19] Pete Cannistraci John Liu, Anant Kadiyala. Enhancing supply chains with the transparency and security of distributed ledger technology. https://www2.deloitte.com/content/dam/Deloitte/global/Documents/Technology/gx-tech-oracle-blockchain-cover-2019.pdf, 2019.

[20] Sunny King and Scott Nadal. Ppcoin: Peer-to-peer crypto-currency with proof-of-stake. 2012.

[21] Ivan Kot. Smart contract applications, limitations and future outlook. https://www.itransition.com/blog/smart-contract-applications, 2021.

[22] Quantstamp Labs. What is a re-entrancy attack? https://quantstamp.com/blog/what-is-a-re-entrancy-attack, 2019.

[23] Alex Coventry Ari Juels Andrew Miller Lorenz Breidenbach, Christian Cachin. Chainlink off-chain reporting protocol. https://research.chain.link/ocr.pdf, 2021.

[24] Benedict Chan Alex Coventry Steve Ellis Ari Juels Farinaz Koushanfar Andrew Miller Brendan Magauran Daniel Moroz Sergey Nazarov Alexandru Topliceanu Florian Tram'er Fan Zhang Lorenz Breidenbach, Christian Cachin. Chainlink 2.0: Next steps in the evolution of decentralized oracle networks. https://research.chain.link/whitepaper-v2.pdf, 2021.

[25] Fernando P. Méndez. Smart contracts, blockchain and land registry. European Land Registry Asociation (ELRA) General Assembly, 2018.

[26] Amirmohammad Pasdar, Zhongli Dong, and Young Choon Lee. Blockchain oracle design patterns, 2021.

[27] Blaž Podgorelec, Marjan Heričko, and Muhamed Turkanović. State channel as a service based on a distributed and decentralized web. *IEEE Access*, 8:64678–64691, 2020.

[28] Team Rocket, Maofan Yin, Kevin Sekniqi, Robbert van Renesse, and Emin Gün Sirer. Scalable and probabilistic leaderless bft consensus through metastability, 2020.

[29] Qiuyun Shang and Allison Price. A Blockchain-Based Land Titling Project in the Republic of Georgia: Rebuilding Public Trust and Lessons for Future Pilot Projects. *Innovations: Technology, Governance, Globalization*, 12(3-4):72–78, 01 2019.

[30] Sergey Nazarov Steve Ellis, Ari Juels. Chainlink - a decentralized oracle network. https://research.chain.link/whitepaper-v1.pdf, 2018.

[31] Koji Takahashi. Blockchain technology for letters of credit and escrow arrangements. Banking Law Journal pp. 89-103, 2017.

[32] Dan Wang, Jindong Zhao, and Chunxiao Mu. Research on blockchain-based e-bidding system. *Applied Sciences*, 11(9), 2021.

[33] Shuai Wang, Yong Yuan, Xiao Wang, Juanjuan Li, Rui Qin, and Fei-Yue Wang. An overview of smart contract: Architecture, applications, and future trends. In *2018 IEEE Intelligent Vehicles Symposium (IV)*, pages 108–113, June 2018.

[34] Neo C.K. Yiu. An overview of forks and coordination in blockchain development. https://arxiv.org/pdf/2102.10006.pdf, 2021.

Appendices

Appendix A

Multi-signature Contract

A.1 Multi-signature Proposal Workflow

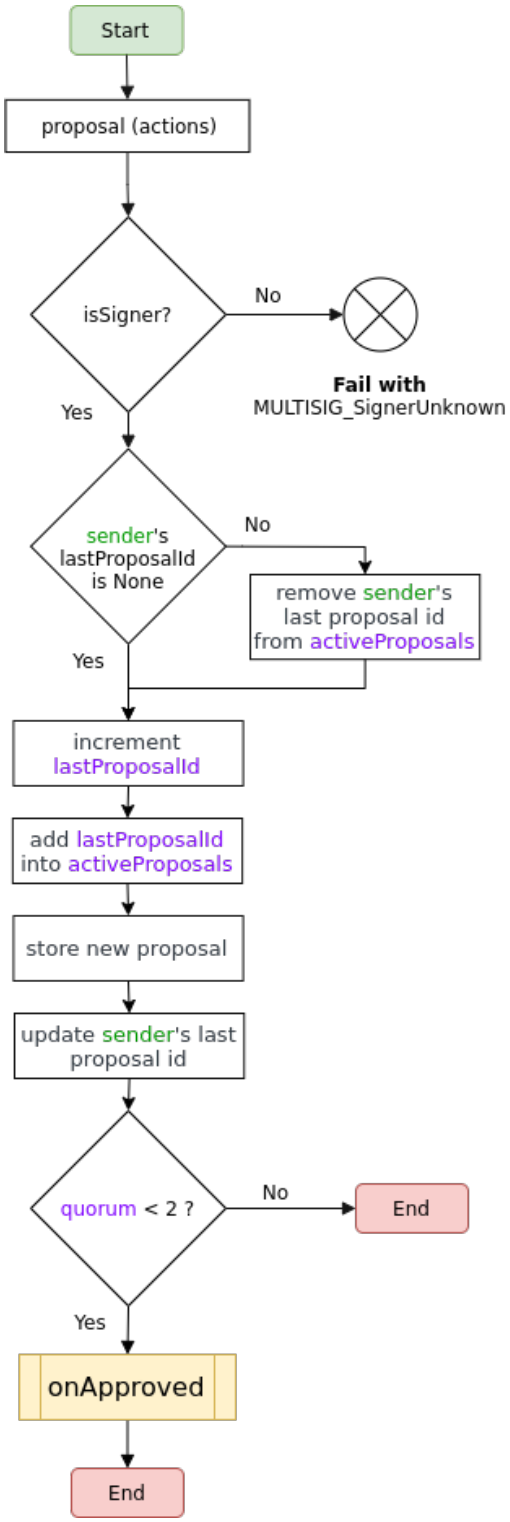

Figure A.1: Multi-signature proposal workflow.

A.2 Multi-signature Endorsement Workflow

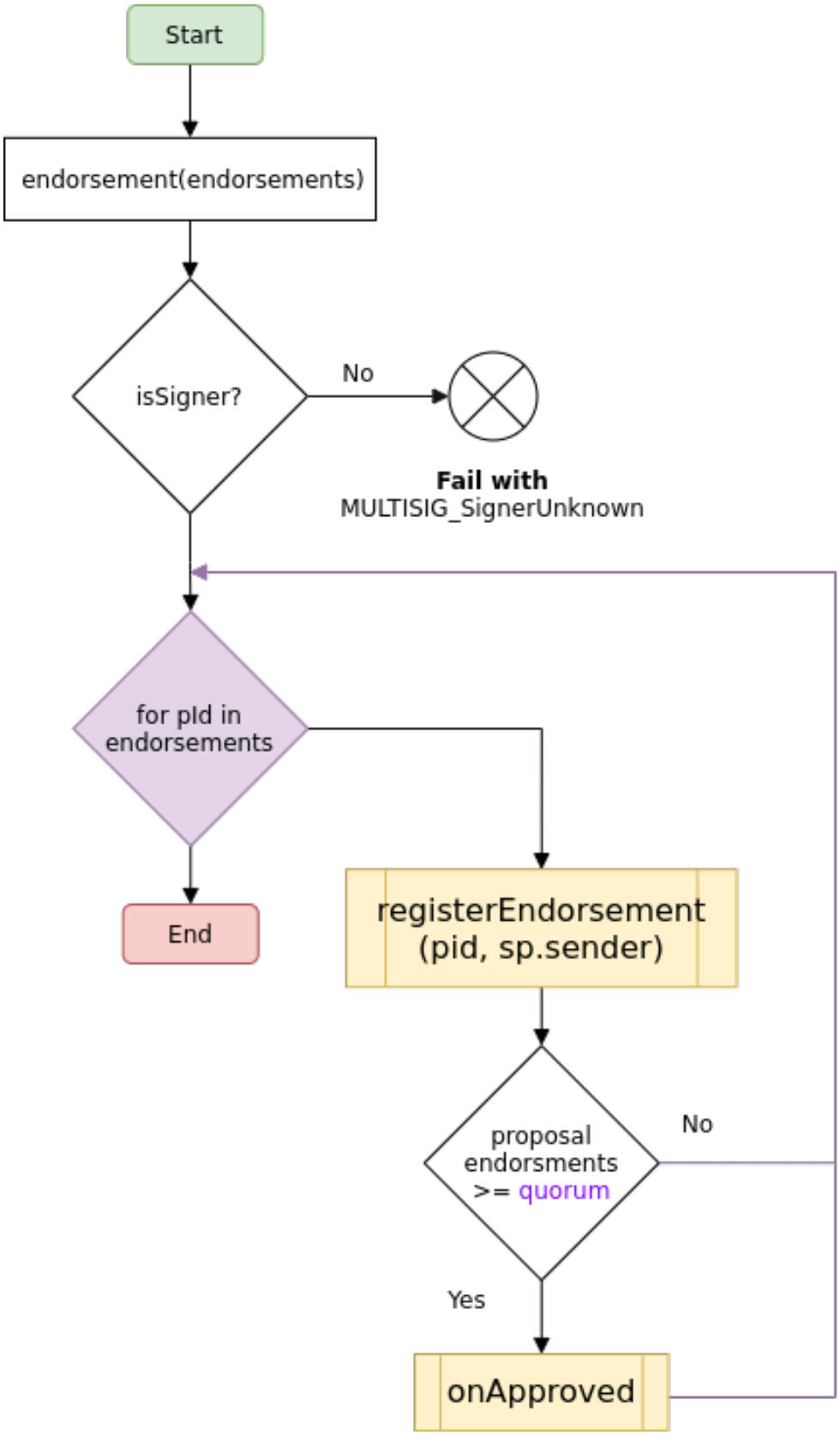

Figure A.2: Multi-signature endorsement workflow.

A.3 Multi-signature Aggregated Proposal Workflow

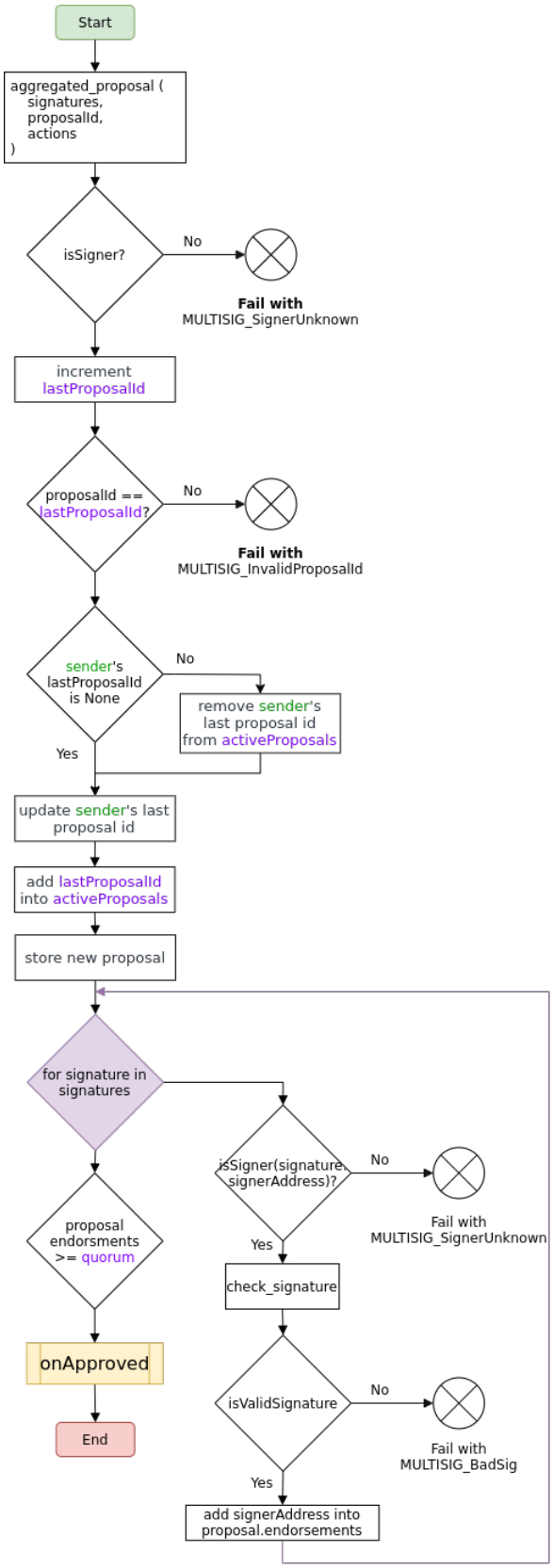

Figure A.3: Multi-signature aggregated proposal workflow.

A.4 Multi-signature Aggregated Endorsement Workflow

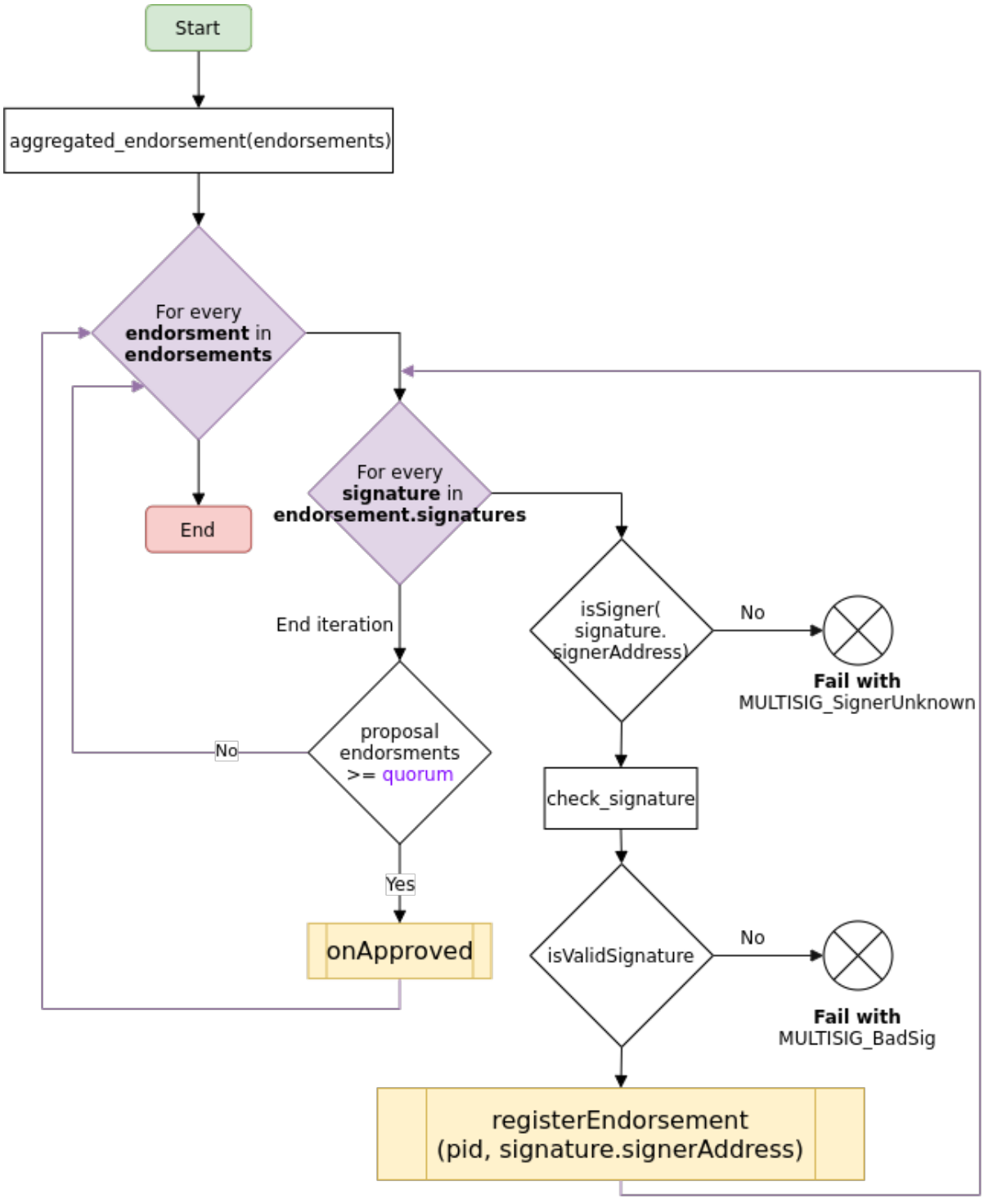

Figure A.4: Multi-signature aggregated endorsement workflow.

A.5 Multi-signature Proposal Cancellation Workflow

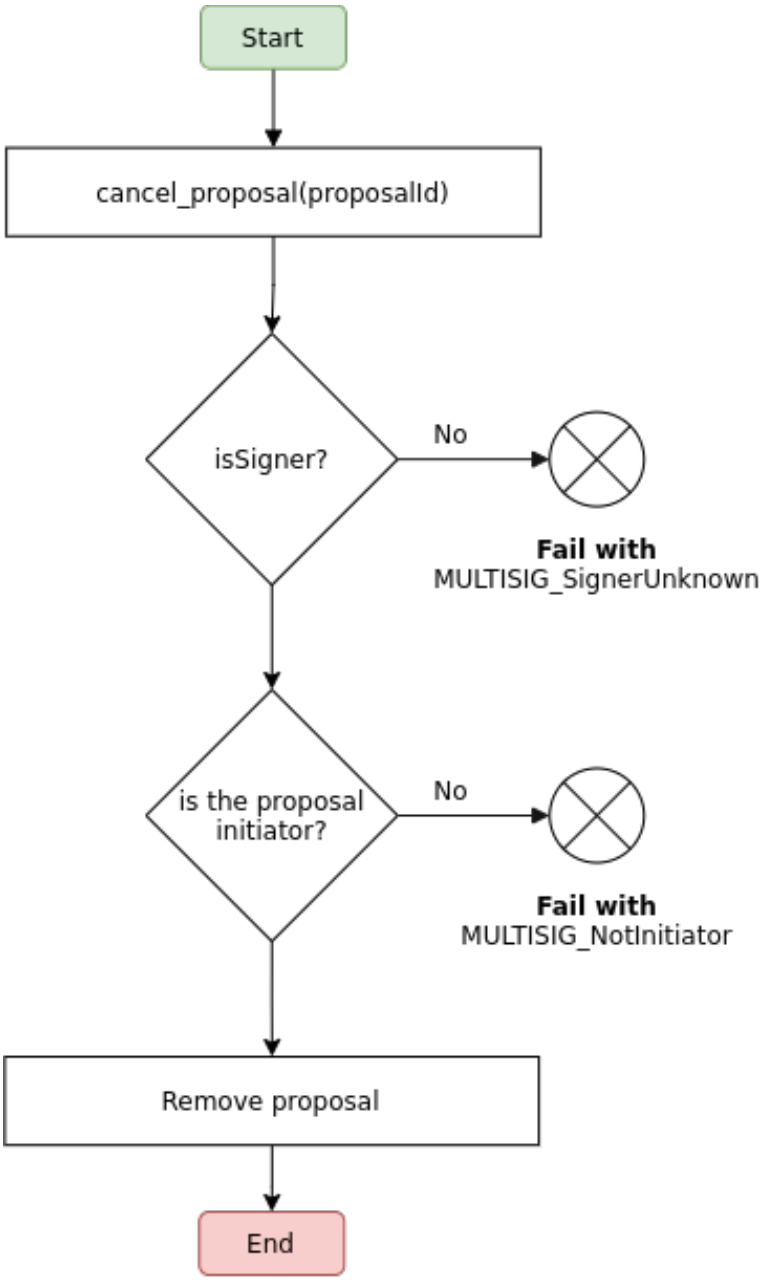

Figure A.5: Multi-signature proposal cancellation workflow.

Appendix B

Price-Feed Contracts

B.1 Price Submission Control-flow Graph

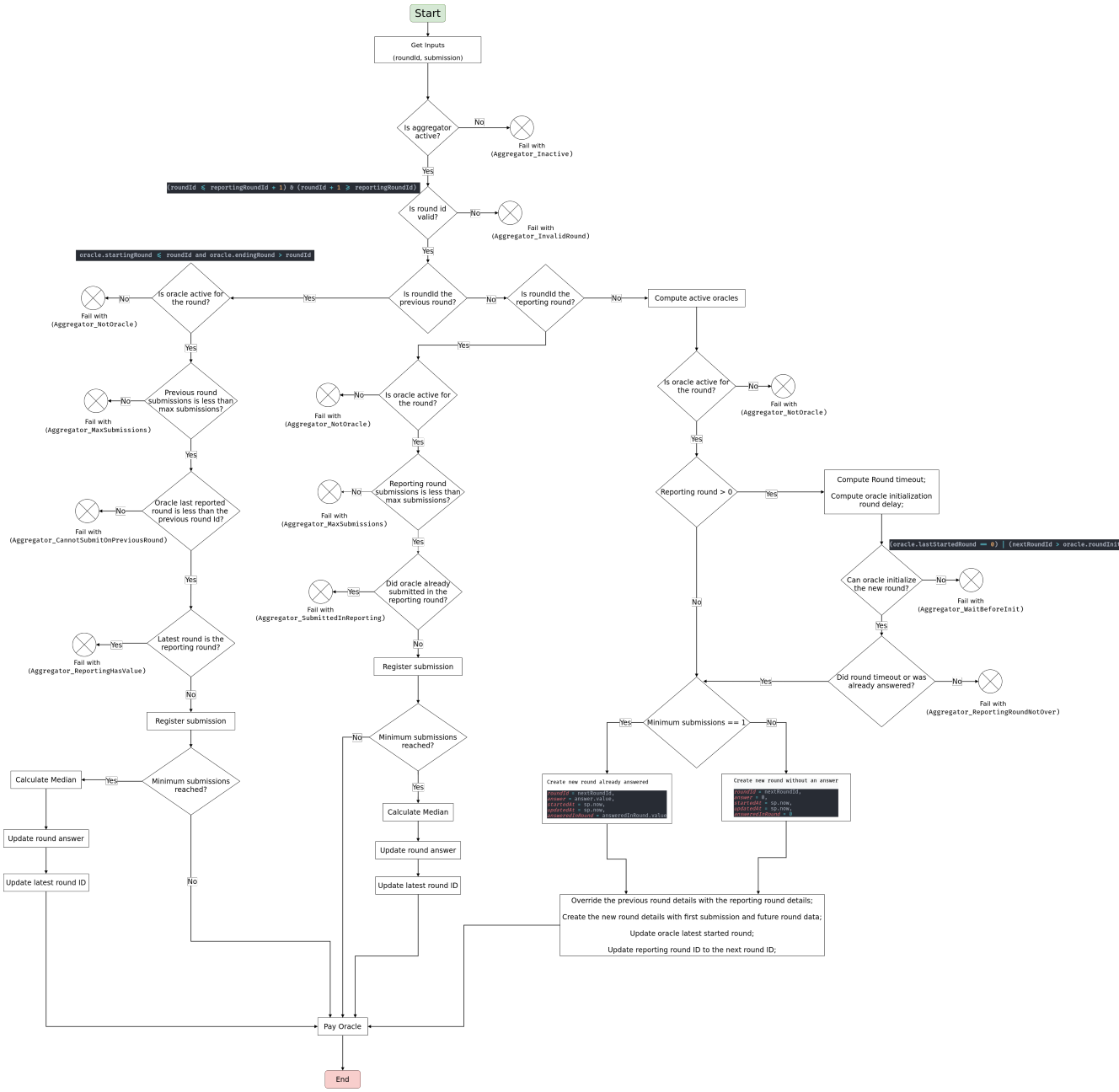

Figure B.1: Price submission control-flow graph.

B.2 Proxied Request Sequence Diagram

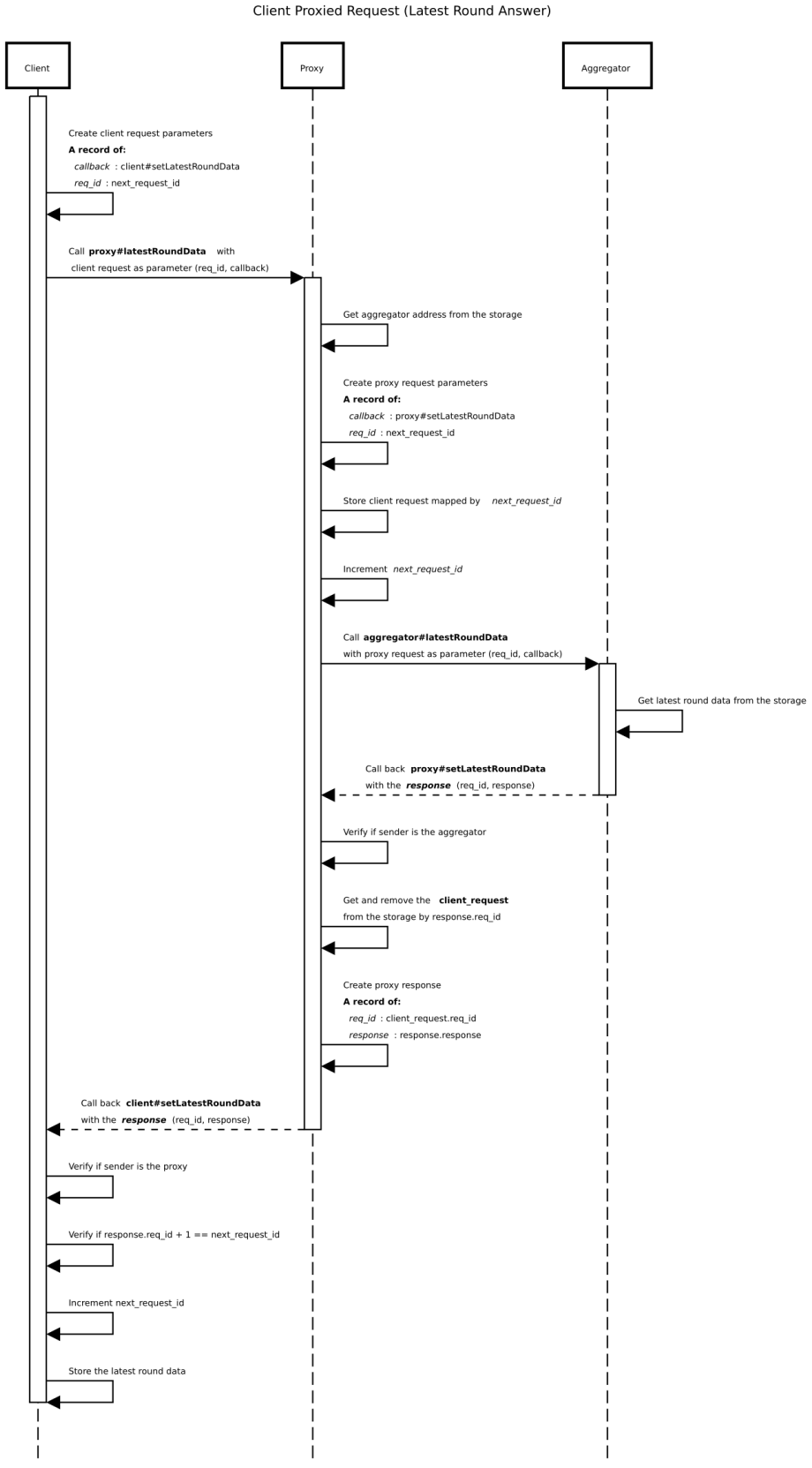

Figure B.2: Sequence diagram that illustrates the proxy workflow.