Cross-Blockchain Smart Contracts

Invoking Smart Contracts Across Blockchains

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

Markus Nissl, BSc
Matrikelnummer 01525567

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Associate Prof. Dr.-Ing. Stefan Schulte

Wien, 20. Juni 2019

Markus Nissl Stefan Schulte

FAKULTÄT FÜR !NFORMATIK

Faculty of Informatics

Cross-Blockchain Smart Contracts

Invoking Smart Contracts Across Blockchains

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

Markus Nissl, BSc
Registration Number 01525567

to the Faculty of Informatics

at the TU Wien

Advisor: Associate Prof. Dr.-Ing. Stefan Schulte

Vienna, 20th June, 2019

Markus Nissl — Stefan Schulte

Erklärung zur Verfassung der Arbeit

Markus Nissl, BSc

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

Wien, 20. Juni 2019

Markus Nissl

Acknowledgements

First, I would like to thank my advisor Stefan Schulte for his support and his fast and constructive feedback on this thesis.

I also like to thank Michael Borkowski for his technical input and comments on this thesis.

Finally, I have to thank my family for their support during my studies.

Kurzfassung

Seit 2017 haben Kryptowährungen große Beachtung gefunden. Pro Monat werden ungefähr 50 neue Kryptowährungen auf Vergleichsseiten gelistet. Diese Währungen werden dabei mittels Transaktionen in verteilten Bestandsbüchern, sogenannten Blockchains, verwaltet.

Die Blockchains werden allerdings nicht nur für „digitales Geld“ verwendet, sondern bieten unter anderem durch codebasierte Verträge, sogenannte Smart Contracts, die Möglichkeit nahezu beliebige Dienste dezentral auszuführen. Mögliche Anwendungen umfassen die Bereiche Gesundheitswesen, Vermögens- und Supply Chain Management.

Da sich die Blockchains in der Funktionalität voneinander unterscheiden, z. B. durch die Unterstützung von Smart Contracts oder durch Privatsphäre-Funktionen, eignen sich bestimme Blockchains für manche Anwendungsgebiete besser. Vor allem im Bereich der Industrie, z. B. im Supply Chain Management, kann daher der Fall eintreten, dass Unternehmen unterschiedliche Blockchains im Einsatz haben. Eine Kommunikation zwischen den Smart Contracts ist jedoch nicht oder nur über Umwege möglich, da Smart Contracts den Anwendungsbereich der jeweiligen Blockchain nicht verlassen können ohne die manipulationssichere Eigenschaft von Blockchains zu verlieren, da die Transaktionen in einer unsicheren Umgebung, z. B. per Hand, übertragen werden müssen.

Daher wird im Zuge dieser Diplomarbeit ein Framework entwickelt, das die Interoperabilität zwischen unterschiedlichen Blockchains erhöht. Es werden dabei sogenannte Cross-Blockchain-Calls eingeführt, die einen Aufruf eines Smart Contracts auf einer anderen Blockchain ermöglichen. Das Framework nimmt dabei Rücksicht auf die Nebenläufigkeit der Blockchains, die „eventual consistency“ von Transaktionen, die Bezahlung von Transaktionsgebühren sowie die korrekte Ausführung der Cross-Blockchain-Calls. Weiters werden die Fortführung einer Transaktion nach Erhalt des Ergebnisses eines Cross-Blockchain-Calls und der rekursive Aufruf über mehrere Blockchains ermöglicht. Der implementierte Prototyp unterstützt dabei alle Blockchains, die auf der Ethereum Virtual Machine basieren sowie den Solidity Compiler verwenden (z. B. Ethereum, Ethereum Classic oder Tron) und kann um weitere Blockchains erweitert werden.

Die Evaluierung hat gezeigt, dass der Prototyp die Transaktionskosten um das 40-fache erhöht und je nach Konfiguration 5-7-mal länger benötigt als das manuelle Importieren eines Transaktionsresultates. Allerdings wird durch die Verwendung des Frameworks der manuelle Aufwand des Aufrufers verringert und der Importwert von unabhängigen Stellen überprüft.

Abstract

At least since 2017, cryptocurrencies got significant attention. About 50 new cryptocurrencies are listed on comparison pages per month. These currencies are managed by transactions in distributed ledgers, so-called blockchains.

However, the blockchains are not only used for "digital money", but also offer the possibility of code-based contracts, so-called smart contracts, that execute services decentralized. Possible applications include health care, asset management and supply chain management.

Since the functionalities of blockchains differ, for example by supporting smart contracts or privacy functions, certain blockchains are better suited for some application areas. Especially in industry, for example in supply chain management, it can happen that companies use different blockchains. As a result, communication between the smart contracts is not or only indirectly possible because smart contracts cannot leave the scope of the respective blockchain without losing the tamper-proof property of blockchains, since the transactions are transferred in an insecure environment, e.g. manually.

As part of this thesis, a framework is developed that increases the interoperability between different blockchains. Therefore, so-called cross-blockchain calls are introduced, which allow to call a smart contract on another blockchain. The framework takes into account the concurrency of the blockchains, the eventual consistency of transactions, the payment of transaction fees and the correct execution of the cross-blockchain calls. In addition, it is possible to continue the transaction after receiving the result of a cross-blockchain call and to recursively call smart contracts across blockchains. The implemented prototype supports all blockchains that are based on the Ethereum Virtual Machine and use the Solidity compiler (e.g., Ethereum, Ethereum Classic and Tron), but can be extended with additional blockchains.

The evaluation has shown that the prototype increases the transaction cost by a factor of 40 and, depending on the configuration, takes a factor of 5-7 longer than importing the transaction result manually. However, the manual effort of the caller is reduced and the import is verified by independent parties.

Contents

CHAPTER 1

Introduction

At least since the hype in 2017, cryptocurrencies got significant attention by the industry and the research community. Since then, many different proposals for novel blockchain technologies have been presented. While Bitcoin [Nak08], the first and in the public most famous cryptocurrency has only limited feature support, new blockchain approaches which provide more sophisticated functionalities arise continually.

Alone CoinMarketCap[1] has more than 2100 cryptocurrencies (of which around 800 coins) listed in March 2019, and the number continues to rise. In February 2019, CoinMarketCap listed about 50 new cryptocurrencies[2], with the actual number of assets created being higher, as asset listing requires trading on at least two exchanges supported by CoinMarketCap[3].

Many of these added coins are just tokens or a variation of an older one, but some of them introduce novel concepts. In general, technology advances can be grouped into the following categories (based on [BRMS18b]):

- Improving the cryptocurrency itself, e.g., adding multi-signature support in Bitcoin.
- Creating spin-offs of cryptocurrencies, e.g., Bitcoin by adapting hashing algorithms (Litecoin [Lit]), block creation rules (Bitcoin Cash [bitb]) or adding features, like privacy-related enhancements (Dash [DD18], Zcash [BSCG+14]).
- Using the cryptocurrency technology as a protocol layer and adding new layers on top of it, like OmniLayer [omn] or the Lightning Network [PD16] for Bitcoin.
- Creating new cryptocurrencies with novel concepts, such as smart contracts, which are well known, e.g., from the Ethereum [Ethc] platform.

[1] `https://coinmarketcap.com/all/views/all/`, accessed 2019-03-19
[2] `https://coinmarketcap.com/new/`, accessed 2019-03-19
[3] `https://coinmarketcap.com/methodology/#listings-criteria`, accessed 2019-03-19

As a developer and researcher it is still difficult to track the progress of each blockchain technology and to choose the promising technologies to build on. Different research groups use different blockchain technologies, users are not sure which one is right for holding assets and developers have the agony of choice to decide which one should be used for their next project. This leads to a fragmented blockchain world.

Marketplaces and exchanges provide for users a money exchange service between cryptocurrencies. These providers act as a third-party service and provide a more or less trustful man-in-the-middle identity for coin exchange, where parties are sending coins in one currency and are receiving coins in a different one. Indeed, these services present a risk for security attacks, as the famous Mt. Gox hack[4] has shown, or other events, such as the death of the owner of an exchange service[5].

Therefore, atomic swap protocols [Her18] were developed, which provide a peer-to-peer cryptocurrency exchange without using a trusted third party. However, even with atomic swaps, the problem of fragmented blockchains is not solved [BRMS18a]. In general, transactions made in one blockchain never leave the ecosystem of the particular blockchain. Therefore, the following actions can be only carried out within a single blockchain [BFS+18]:

- Sending coins between various participants
- Calling smart contracts saved in the blockchain
- Accessing data stored in the blockchain

This has the effect that users cannot move their assets between different blockchains, and developers cannot use libraries written for one blockchain without conversion on a different blockchain. In addition, the conversion step gets difficult, when libraries themselves have dependencies or functional requirements that are only supported by the primary blockchain. Furthermore, researchers cannot use current work in their preferred blockchain, since novel concepts are targeted for a particular blockchain technology.

To mitigate the problem of cross-blockchain interaction, more recent work has introduced methods to transfer tokens across several blockchains [Aut18, BRMS18b]. While these methods focus on token transfers, i.e., sending coins between various participants in different blockchains, the other topics mentioned in the list above are still open and of important interest.

To the best of our knowledge, finding a way to interact with a smart contract on a different blockchain has not been regarded so far, although this would lead to various possibilities in industry and research. For example, the library and function dependency

[4]`https://blockonomi.com/mt-gox-hack/`, accessed 2019-03-19

[5]`https://www.theguardian.com/technology/2019/feb/04/quadrigacx-canada-cryptocurrency-exchange-locked-gerald-cotten`, accessed 2019-03-19

issue would be solved by invoking a smart contract of one blockchain by a different smart contract on a different blockchain. A concrete usage scenario of a fictional example is described in Section 1.1.

Therefore, the main focus of this thesis is to design and develop methods for calling smart contracts across blockchains.

1.1 Motivational Scenario

There are several use cases for calling smart contracts on a different blockchain. This section describes in detail a scenario in the area of supply chain management. It shows the intuition behind invoking smart contracts between blockchains. The requirements are then summarized.

1.1.1 Description

Assume a company called "CarTech" is a manufacturer of vehicle components. This company needs for production several materials, such as screws or steel, from vendors and delivers its produced parts, for example an engine, to the car manufacturer. For this process, a supply chain is usually set up, in which each participating company is subject to rules. One of these rules usually includes delivery conditions and a penalty fee if they are violated. However, detecting the guilty party is often more difficult than expected. Without having an immutable tracking system, companies could, for example, adjust the date of a good arrival or the date of shipment of their produced commodities.

In order to ensure immutable data tracking, the use of blockchains is suitable. Each company publishes its data values in real time on a blockchain of its choice. These include vendors, clients and delivery services. This way, continuous tracking data is published and a complete record is available.

CarTech is interesting in analyzing the published data to identify events for abnormalities. For that, the company has created a smart contract, which is called twice a day to retrieve up-to-date data from the blockchains. This contract interacts with the smart contracts of its business partners by invoking methods for obtaining published values.

Someday, a vendor called "CrazyParts" develops a new material that is lighter and more stable than the previously used steel. CarTech, as a leading car part manufacturer, is interested in using this material for its product parts. Unfortunately, CrazyParts has chosen another blockchain to save its data values before entering into partnership with CarTech. Thus, the smart contract of CarTech cannot interact with the supply chain data of CrazyParts, since there is currently no such interaction between blockchain technologies. The development effort of CarTech's smart contract is thereby lost, since the automatic abnormality system is ignoring the new vendor.

Therefore, CarTech has to download and save the other blockchain locally, then either manually check for abnormalities or publish the values in the blockchain used by CarTech, so that the developed smart contract can scan the values there. This is increasing the local storage capacity, computational power and cost. While this may be a potential solution for CarTech for one partner company, the company is not interested in taking this step for many new vendors.

Therefore, CarTech is looking for a way to invoke smart contracts on a different blockchain inside their own smart contract logic.

1.1.2 Requirements

Based on the motivational scenario, the following requirements can be extracted:

1. **Invoking smart contracts on a different blockchain.** This requirement is the main objective of the thesis. It should be possible to call smart contracts on another blockchain with varying parameters and return values.

2. **Scalability.** The solution should be scalable to multiple parties, smart contracts and blockchains.

3. **Security.** Invoking smart contracts requires passing of parameters and return types. To ensure that correct data is processed, the integrity of the values must be preserved. For this, a method for verifying the integrity of parameters and return types is required.

4. **Consistency.** Blockchains are using a concept of eventual consistency. A block is considered to be more consistent with higher probability, as more blocks follow the current block. This has to be taken into account when receiving values from other blockchains.

1.2 Research Questions

Following the motivational scenario from the previous section, multiple research questions have to be addressed. The following list summarizes the research questions of this thesis:

1. **Which blockchain technologies are suitable candidates for implementing cross-blockchain smart contract calls? Which methods for interacting between blockchains exist? Can they be used as a basis for invoking smart contracts?**
   This task builds heavily on literature search. Thereby, different approaches are discussed based on the requirements. The research is focusing on the following concepts:

   a) Which technological features are supported by well-known cryptocurrencies?

   b) How are atomic swap protocols implemented?

c) What is the current status and behavior of cross-blockchain token transfer protocols?

2. **How can smart contracts be called across blockchains?**
   Inspired by related work, a method for invoking smart contracts across blockchains is designed, which is addressing the mentioned requirements.

3. **How does the prototype perform in terms of cost and performance?**
   The created prototype is evaluated by empirical evaluation in test networks. The obtained measurements will be used to calculate prices to estimate the trade-off by using this architecture.

1.3 Methodology

The methodological approach of this thesis consists of the following phases:

1. **Literature research.** In the first part, required background information about blockchain technologies is gathered. Of particular interest are recent work on atomic swaps and cross-platform token transfers, as well as supported features of various blockchain technologies. The collected information is used as input for the design process. Thereby, it helps finding a basic concept for calling a method on a different blockchain and selecting a blockchain technology for the prototype.

2. **Design.** Based on literature work, a method for invoking smart contracts across blockchains is developed. For this, several approaches will be discussed. Among other, consistency issues related to the concurrent environment of multiple blockchain technologies are addressed. An important part is the modeling of transaction flows between different blockchains including parameter and result propagation between the interacting blockchains. Moreover, based on the chosen method, a conceptual architecture for cross-blockchain interaction is designed. In addition to a functional design, it is of great interest to maintain the current level of security of blockchains and to detect possible introduced illegal transaction flows such as double execution (spending) or integrity violation.

3. **Implementation.** The designed architecture is used to implement a prototype. This prototype is a proof-of-concept which demonstrates the invocation of smart-contracts across block chains.

4. **Evaluation.** The prototype is evaluated with different settings. These contain a varying number of parallel transactions, parameters, return values and smart contracts. The results are then analyzed with different metrics, such as execution time and cost, which can be used to improve the prototype as well as to evaluate the proposed solution. Since the evaluation is part of the implementation process, these steps are performed iteratively.

1.4 Structure

The remainder of this work is organized as follows:

- **Chapter 2** summarizes key concepts of blockchain technologies needed for the following chapters, introduces smart contracts, and outlines important characteristics of top-ranked blockchain technologies.
- **Chapter 3** discusses the current state of the art in cross-blockchain interoperability. The focus is on related work regarding atomic swap protocols and cross-blockchain tokens.
- **Chapter 4** first selects a blockchain technology for implementing a prototype and sets up functional use cases for the design. The remainder of this chapter is focusing on the method design and the architecture.
- **Chapter 5** describes the implementation of a prototype for cross-blockchain smart contracts.
- **Chapter 6** evaluates the results from the previous chapter by quantitative measurements and discusses these results.
- **Chapter 7** concludes this thesis by summarizing the proposed solution and outlining plans for future work.

CHAPTER 2

Background

This chapter summarizes basic concepts of blockchain technologies used in this thesis. First, the elementary concepts of blockchains are briefly presented, followed by a detailed explanation of smart contracts. The chapter concludes with a brief description of various blockchain technologies.

2.1 Blockchain Fundamentals

This section will quickly review the basics of cryptocurrencies.

2.1.1 Cryptographic Concepts

Hash Functions. A hash function (H) transforms a variable-length input into a fixed-length output. This mean that for any value v it is easy to compute $H(v)$, but it is infeasible to discover a $v' \neq v$, so that $H(v') = H(v)$ [Her19]. Different blockchain technologies use different hash functions, like SHA-256d [Nak08] or Scrypt [Lit].

> **Example 2.1** *Assume the input "abc". The SHA-256 hash would be "ba7816bf8f01cfea414140de5dae2223b00361a396177a9cb410ff61f20015ad"*

Asymmetric Cryptography. Asymmetric cryptography describes the concept of using public and private keys to encrypt and decrypt values. Which key is used for encryption depends on the usage [Her19]:

- Confidentiality: If the public key is used for encryption, the message can only be read by the owner of the private key.
- Digital Signature: If the private key is used for encryption, anyone can verify that the message was encrypted by the owner of the private key.

Example 2.2 *Alice wants to sign a file to prove that the file was issued by her. Therefore, she calculates the hash value of the file, which she encrypts with her private key (she signs the document). Afterwards, she adds the signature to the file.*

Bob wants to check if the document was created by Alice. Therefore, he decrypts the signature attached by Alice and calculates the hash value of the file with the same function as Alice. Then he compares the two values if they are the same.

2.1.2 Transactions

A transaction is a contract for an action. Possible actions include token transfers and invoking smart contracts (see Section 2.2). Each transaction must be authorized by the account holder so that anyone can verify that the transaction was made on behalf of the holder. This is achieved through the use of digital signatures. To prevent reuse of the signed document, each transaction has a concept similar to a serial number to identify the uniqueness of the document. For example, Bitcoin uses unspent transaction outputs (UTXOs). Thereby, received transactions can only be used once to generate new transactions [Her19, TS16].

Example 2.3 *Assume following action: Alice wants to send Bob a coin. For this reason, Alice has to generate a signed transaction with the information "Use transaction #1234 with a value of 5 coins. Pay 1 coin to Bob's address, 4 coins to Alice's address."*

2.1.3 Blockchain

The blockchain is a distributed ledger that contains all published transactions in a sorted order. Each transaction stores the hash value of the previous transaction entry and points to the transaction. This builds a chain of transactions (single linked list) where the hash has the task of making the chain tamper-proof.

In fact, the transactions in the blockchain are grouped into blocks for efficiency, while the transactions themselves are stored in a separate data structure within the block (see Section 2.1.4 on Merkle Trees). Therefore, the link with its hash is applied only to the blocks, which is visualized in Figure 2.1. Modifying any value in the chain changes the calculated hash in the next block, indicating an anomaly in the blockchain [Her19, TS16].

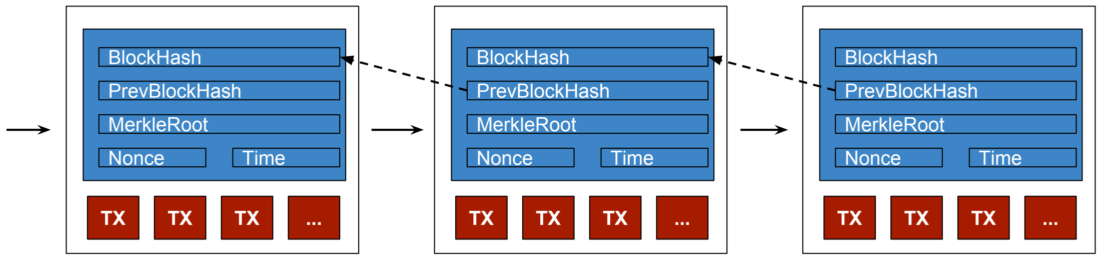


Figure 2.1: Visualization of a Simplified Blockchain [TS16].

2.1.4 Merkle Trees

Transactions are stored in blockchains (at least in Bitcoin [Nak08] and Ethereum [Ethc]) in so-called Merkle trees. Thereby, each transaction is placed at a leaf in the tree. The parent nodes contain hashes of their child nodes. Therefore, a branch in the tree can be easily verified to the end, without the need for other branches. Only the siblings along the path are required, since any modification would affect the root hash of the tree. Figure 2.2 shows a graph representing the path (red) to transaction B3 with the siblings (green) required for verification [Aut19].

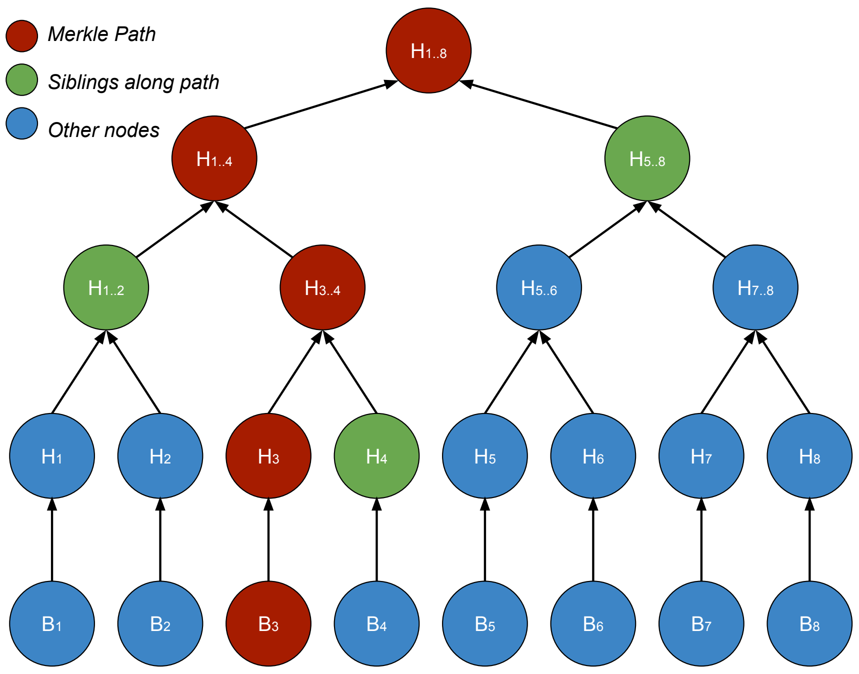


Figure 2.2: Merkle Path [Aut19]

2.1.5 Consensus Protocol

The order of the blocks is determined with the consensus protocol. It is a protocol between multiple parties (so-called miners), which try to append their generated block to the ledger such that all honest parties agree on the selected block [Her19, TS16].

There exist many variations to reach consensus among honest users. A few protocols, which are often used in the world of cryptocurrencies are shortly explained in Section 2.1.6. In general, a consensus protocol is not more than a agreement on a set of rules that describe the validity of a transaction, of a block and how the information is propagated between the participating miners [Nak08].

2.1.6 Consensus Algorithms

Several approaches for reaching consensus in blockchains have been proposed. This section summarizes the most common ones in one paragraph each.

Proof of Work (PoW). Participants of the blockchain, so-called miners, have to solve a complex puzzle to include a block in the blockchain. This puzzle requires finding a nonce so that the hash of the block begins with a certain number of zeros. The number of zeros is determined by a variable called the difficulty, which is dynamically adjusted depending on the total computing power. Since a hash can be recalculated, when the nonce is known, miners can easily verify the correctness of the nonce. For example, this algorithm is used in Bitcoin. However, calculating the nonce requires a lot of energy and time. Therefore, alternative approaches have been developed [Sun18].

Proof of Stake (PoS). The creator of the next block is selected by a pseudorandom function. To participate, a node has to build up a stake, which will be lost through malicious behavior. There exist several variations of the consensus protocol that contain different parameters to influence the likelihood of selection, for example the age of the stake [Sun18].

delegated Proof of Stake (dPoS). It is an adaption of PoS where any token holder can vote for witnesses who create and validate blocks, as well as delegates who can change blockchain parameters, such as transaction sizes or block intervals. Therefore, there are only a small amount of witnesses and thus the protocol is much faster, but more centralized. The selection of the leader for creating the next block is deterministic [Sun18].

Proof of Authority (PoA). The blocks are validated by approved accounts that may lose the reputation by behaving malicious. Since the identity of the validator needs to be verified in the real world, it becomes much more difficult to be chosen as validator. Therefore, this protocol is controlled very centrally [Sun18].

2.1.7 Blockchain Forks

Sometimes, two parties practically simultaneously propose a block. Consequently, the participating parties receive different blocks on which to build: A fork happened. The forks may join together after a new block has been proposed, as honest parties always follow the longest chain [Her19]. The blocks, which are not part of the longest chain anymore are called stale blocks.

There exist also intended forks of the developers of a cryptocurrency: soft forks and hard forks. While hard forks are not backward compatible because old nodes do not accept blocks created by new nodes, soft forks prohibit only blocks created by old nodes because of restrictions in the validation rules. However, both types of nodes accept the blocks of the new nodes, so the longest chain does not diverge [ZSJ$^+$19].

2.1.8 Rewards and Fees

To acquire miners for the blockchain, a kind of reward is necessary. Bitcoin has introduced two types of rewards: block mining rewards and transaction fees. Block mining rewards are paid to the miner who is able to successfully append the next block to the longest blockchain, while transaction fees are paid by participants who wish to add a transaction to the blockchain. Depending on the amount of the fee, the transaction may sooner or later be considered by the miners. The fee is calculated in Bitcoin by the difference from outgoing and incoming transactions [TS16].

Example 2.4 *Carole is mining a block. The current block reward is 12.5 coins. The block contains a transaction from Alice. She uses an unspent transaction of 5 coins and sends 2.5 coins to Bob. Carole can now collect 15 coins for mining the block (12.5 block reward + 2.5 transaction fee).*

2.1.9 Security Issues

This subsection summarizes commonly discussed cryptocurrency attack vectors.

51% Attack. If one party owns at least 51% of the mining resources, it has more resources than the rest of the participating miners. This would have a negative impact on the network, if the party behaves maliciously. Transactions could be held back for being added to the blockchain and already published blocks can be rewritten due to the high amount of resources by creating a chain longer than the current one. This would destroy the trustworthiness of the currency [TS16].

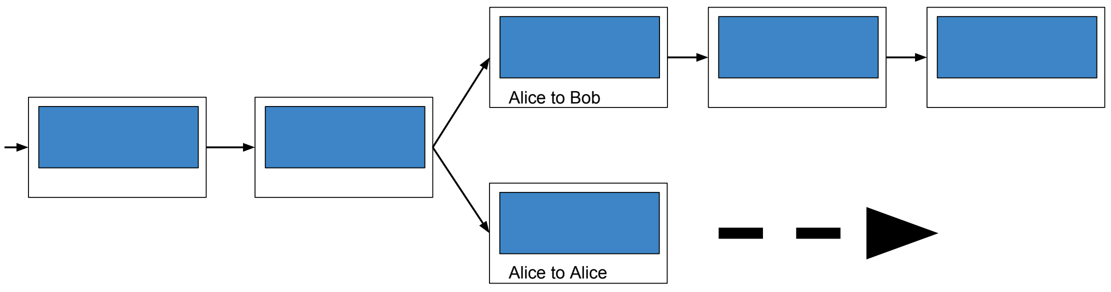


Figure 2.3: Double Spending Attack [TS16].

Double Spending Attack. This attack describes the flaw of spending a coin multiple times. For example, in Bitcoin, this problem is solved by the verification step of the consensus protocol, where each transaction has to be checked for double spending attacks by verifying that the UTXO has not already been used. A different form of a double spending attack would be in form of the 51% attack, where the blocks can be rewritten by creating the longest chain, which is also shown in Figure 2.3. Thereby, the recipient already assumes that the transaction has been successfully verified (the longest blockchain contains the transaction) and fulfills its part of the agreement [TS16].

Example 2.5 *Alice pays 5 coins to Bob for a concert ticket. Bob sends the ticket to Alice after 6 blocks have been built on the block containing the transaction (very likely that the transaction is valid). However, Alice forks the blockchain in front of the block containing her transaction and alters her transaction so that Alice sends the 5 coins to herself. Since she owns over 50% of the computation power, Alice can generate the longest chain and publishes it to the other miners. Honest miners are now considering her chain as the longest and start mining on this one. The previous chain is "gone". As a result, Bob has received no coins and lost his concert ticket.*

Sybil Attack. If a network decides only on the majority of votes, a party can create a lot of fake identities to win the vote, as long as there exist no possibility to verify the parties. For example, this is the case in cryptocurrencies, where any number of public/private key pairs can be generated by one party. To mitigate the problem, the voting system is tied to resources (e.g., computing power), which are limited. As a result, the creation of fake identities leads to a distribution of the resources, which reduces the individual voting power of an identity [TS16].

2.2 Smart Contracts

Ethereum is one of the first and most widely used cryptocurrency to support quasi Turing-complete smart contracts [Ethc, BMM$^+$20, TS16]. The following subsections introduce smart contracts by discussing the concepts of scripting in blockchain technologies, necessary Ethereum basics for understanding smart contracts, potential applications and vulnerabilities.

2.2.1 The Beginning of Blockchain Scripting

Bitcoin already supports a weak version of smart contracts, although the functions are very limited. For a better understanding, it is necessary to break down the simplified view of transactions and look at them at a more sophisticated level.

Transactions contain not only the address of the recipient, but are made up of several operation codes. These operations are executed in a simple stack-based programming language and are Turing-incomplete. The script of sending coins to a public address looks as follows: Take a signature as input, verify it with the transaction and the owner of the UTXO and return 1 if the verification is successful. The steps are shown in Figure 2.4. Of course, there exist more complex scripts, such as multi-signatures, where more than one key is required to authorize a transaction (for example, 2 out of 2 or 2 out of 3) [Ethc].

However, since no loops are allowed, the scripting language of Bitcoin does not support all features. Bitcoin only tracks if UTXO are spent entirety, so no additional state is

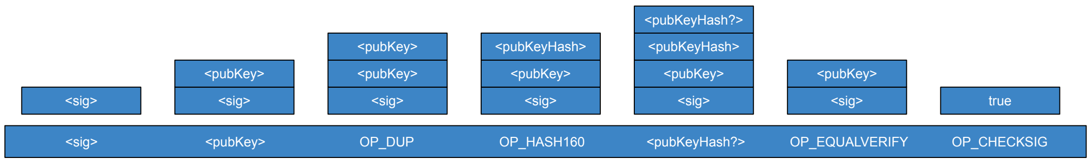


Figure 2.4: Example of a Bitcoin Script Execution [NBF+16].

possible. Therefore, all information has to be saved in the transaction output. Ethereum is using a different attempt for saving the balance of an address and for tracking the ownership status of a coin, which is discussed in the following sections [Ethc].

2.2.2 Ethereum Basics

This section summarizes the basic concepts of Ethereum that have not previously been addressed and are needed for understanding the behavior of smart contracts.

Accounts. In Ethereum, two types of accounts exist: external controlled and contract accounts. Every account contains four fields: a `nonce` for security aspects, an `ether balance` to replace UTXO with a balance state, a `contract code` (used only for contract accounts) and an `account storage`. External accounts can create and receive transactions, while contract accounts must be poked by a transaction or message to get active. Then, contract accounts can send other messages or read and write to their account storage [Ethc].

Transaction prices. A transaction in Ethereum does not cost a static fee like in Bitcoin, but depends on the number of computational steps. Therefore, each transaction includes parameters for controlling the fees. `startgas` describes the maximum number of allowed steps and `gasprice` is a fee, which is paid for each executed step. If `startgas` is reached, the transaction is aborted [Ethc].

Messages. A message is like a transaction, except that it is created by a contract account and not by the initiator of the transaction (external account). Therefore, it has no field for defining the gas price per step. However, the maximum `startgas` of the message can be limited. Messages provide contract creators the possibility to call and interact with other smart contracts and to send money to other accounts [Ethc].

State Transitions. A state transition describes the behavior of changing the state of a blockchain by executing a transaction. It is defined as follows: A state transition accepts a state S and a transaction Tx as input and outputs a new state S' by executing the transaction $((S, Tx) \to S')$. If an error occurs, it depends on the blockchain how it is handled. For example, in Ethereum, the state is reverted if a message runs out of gas during execution. However, fees have to be paid for executing the transactions and parent messages are resumed and not reverted (as long as the message was sent with a lower `startgas` limit than the remaining transaction) [Ethc].

Difference in Blockchain. Since Ethereum needs to store the state of the blockchain (storage, balance, receipts of transactions), each block contains not only the list of transactions (as in Bitcoin), but also the current state of the blockchain in the Merkle tree (in fact, Ethereum uses a more complex version called Merkle Patricia tree) [Ethc].

2.2.3 Ethereum Smart Contracts

The code for Ethereum smart contracts is written in a stack-based, low-level bytecode language called "Ethereum virtual machine (EVM) code". Each byte of the code represents an operation to which a particular instruction sequence applies.

Example 2.6 *Assume the operation ADD. This operation has to pop two values from the stack, calculate the total, and push the result back on the stack. Furthermore, the remaining gas has to be reduced by 1 while the program counter has to be incremented.*

EVM code has access to multiple types of storage: A stack, an infinitely expandable byte array called memory, and the account storage that is not lost after the execution is completed. In addition, smart contracts can access information about the sender, incoming data, and block header data and can return a byte array as output [Ethc].

Ethereum also supports logging functionalities through events. The logs are stored in a special data structure within a block and can only be retrieved by clients and not by the contracts (not even from the contract that created the log entry). Moreover, so-called "delegatecall" functions allow the call of libraries in the context of the caller contract, which means that the library uses the storage of the caller [Ethb].

For completion, the Application Binary Interface (ABI) is a specification that contains metadata about the functions used in a contract and can be used in a static environment for invoking other contracts. A code (Listing 2.1) that visualizes some of the features of smart contracts is written in Solidity, a programming language for Ethereum similar to JavaScript, with a corresponding ABI-File (Listing 2.2) [Ethb].

2.2.4 Applications Using Smart Contracts.

There exist several ways for using smart contracts. In addition to the motivational scenario presented in the introduction, smart contracts can be used for creating financial applications, such as sub-currencies, wills, employment contracts or non-financial applications like systems for online voting, health care or asset digitization (ownership tracking of physical assets) [Ethc, Ash18].

Listing 2.1: Example Contract Code [Ethb]

```
contract Test {
  bytes32 b;

  constructor() public {
    b = hex"12345678901234567890123456789012";
  }

  event Event(uint indexed a, bytes32 b);
  event Event2(uint indexed a, bytes32 b);

  function foo(uint a) public {
    emit Event(a, b);
  }
}
```

Listing 2.2: Example Contract ABI [Ethb]

```
[{
  "type":"event",
  "inputs": [
    {"name":"a","type":"uint256","indexed":true},
    {"name":"b","type":"bytes32","indexed":false}
  ],
  "name":"Event"
}, {
  "type":"event",
  "inputs": [
    {"name":"a","type":"uint256","indexed":true},
    {"name":"b","type":"bytes32","indexed":false}
  ],
  "name":"Event2"
}, {
  "type":"function",
  "inputs": [{"name":"a","type":"uint256"}],
  "name":"foo",
  "outputs": []
}]
```

ERC20-Tokens. ERC20 is a technical standard on the Ethereum platform. It defines an interface for implementing tokens as a sub-currency in smart contracts, which includes methods for receiving the name, symbol, number of decimals, total token supply and current account balance as well as functions for sending tokens and granting permissions to others for withdrawing tokens from its own account. This standard has the benefit that tools and external platforms can easily interact and list the tokens [VB15].

2.2.5 Smart Contracts and Concurrency Issues

A smart contract can be seen as an object. It has a constructor, a long-lived state (storage) and various functions (methods) that manage the states. Every contract lives in the blockchain. When a miner is generating a block, each transaction is ordered and executed one-by-one (for completion: there are discussions to let them run in parallel, as long as the transactions do not conflict with each other [DGHK17]). Therefore, it seems that a developer does not have to worry about the problems of executing a method in parallel.

Listing 2.3: Vulnerable DAO code [Her19]

```
function withdraw(unit amount) {
  client = msg.sender:
  if (balance[ client ] >=amount} {
    if (client.call.sendMoney(amount)) {
      balance[ client ] -= amount;
    }
  }
}
```

Listing 2.4: DAO Hack [Her19]

```
function sendMoney(unit amount){
  msg.sender.withdraw(amount)
}
```

However, as described above, smart contracts can directly invoke other functions of themselves or interact with other accounts that may be also smart contracts. Since many developer ignored this feature, many security issues arose. A famous example was the Decentralized Autonomous Organization (DAO) hack. DAO was a crowd-funding platform where investors could vote, which party should receive the money [ABC17]. The vulnerability was a typical re-entrance attack in which the balance of the party was not reduced before the function was called again. In order to receive more money than intended, a hacker only has to call the function again within its receiving function in order to bypass the guard for checking if enough money is left. The concepts of the vulnerability are shown in Listing 2.3, followed by a possible attack in Listing 2.4 [Her19, ABC17].

The DAO hack was not the only issue, there exist also flaws in other contracts [Ban18, BDJS17, kin16]. For example, the official ERC20 specification includes a warning that before changing the debit value to another positive value, the value must be first set to zero, otherwise more money than desired can be debited [VB15].

2.3 Blockchain Technologies

After the introduction of the key concepts of blockchain technologies, this section introduces the top-ranked technologies. Therefore, the top 10 coins are analyzed by market capitalization[1] (with at least $1,000,000,000 market cap).

Bitcoin-based technologies. Bitcoin [Nak08], the first blockchain-based cryptocurrency, is limited in features. It has a scripting language called Script and supports third-party approaches by adding a layer above Bitcoin for transferring different type of assets beside Bitcoin itself, like colored coins [col] or OmniLayer [omn]. Bitcoin uses PoW as a consensus protocol. Since Bitcoin was the first blockchain-based cryptocurrency many forks were made. Bitcoin Cash was created by a hard fork and Bitcoin SV was

[1] `https://coinmarketcap.com/coins/`, ranking from 2019-03-23

again a fork of Bitcoin Cash, which both increased the block size. Moreover, there were also forks in the code-base that modified some algorithms used in Bitcoin. Litecoin [Lit] used a different hash algorithm and reduced the block generation time. Dash [DD18] was again a fork from Litecoin, which added additional privacy features and used masternodes for the consensus algorithm.

Ethereum. As discussed in the previous section, Ethereum [Ethc] is mainly known because of its ability to add smart contracts and its EVM. It is a Turing-complete scripting language and uses currently a PoW consensus protocol, like Bitcoin. However a change to PoS is planned[2]. Ethereum Classic is a hard fork of Ethereum.

Ripple. Ripple [SYB] is a currency using the Ripple network. All coins have been issued in the genesis block, but are distributed monthly and can be bought by users with money. The Ripple network itself also supports additional assets, like fiats or other cryptocurrencies, and uses a custom consensus algorithm in which a voting system decides which transactions are selected in the next round. Thereby, each participant chooses trusted participants to validate their transactions. Since Ripple does not support smart contracts, the currency will not be further discussed in this thesis [CM18, Sha].

Stellar. Stellar [Maz16] is a cryptocurrency technology that has a similar code-base as Ripple and provides a cryptocurrency called Lumen. Like Ripple, it uses a custom consensus protocol, the Stellar Consensus Protocol. While Ripple focuses on money transfers between banks, Stellar creates a network where people can transfer money between currencies. Stellar also supports “smart contracts”, but these have limited functional support, are made for payment agreements and are not Turing-complete[3]. Therefore, this technology (like Ripple) is not discussed further in this thesis.

Eos. Eos [blo18] is a cryptocurrency running on the eos.io blockchain. The blockchain has no transaction fees and supports parallel processing of transactions by dividing a block into cycles, each containing shards that can be executed in parallel. Furthermore, it uses a dPoS consensus protocol and supports smart contracts. These contracts can be written in WebAssembly, which means that developers can write their programs in C++. Since no transaction fees are applied, EOS uses a different model to limit the bandwidth of the application. Depending on the percentage of EOS tokens associated with an account, the accounts receive partial resources (storage, CPU, RAM) on the network. Furthermore, developers of smart contracts have to pay for the resources.

[2]`https://www.mangoresearch.co/ethereum-roadmap-update/`, accessed 2019-03-27
[3]`https://medium.com/goodx/the-magic-of-stellar-smart-contracts-4519d2cb1b03`, accessed 2019-03-23

Tron. Tron [TRO18] is a cryptocurrency technology focusing on the entertainment sector. It was a fork from Ethereum and thus supports smart contracts with similar features. Like Eos, Tron uses the dPoS consensus protocol. However, there are fees for certain operations and frequent use of the blockchain (calculated on a daily basis). Moreover, it uses "buyable" energy for interacting with smart contracts.

Cardano. Cardano [KRDO17, CD17] is a multi-layered cryptocurrency whose concepts are based on peer-reviewed scientific and academic theories. While the first (settlement) layer handles the transactions and uses the native coin ADA, the second (computing) layer is responsible for handling smart contracts. The computing layer supports also other coins. Cardano uses the PoS consensus protocol [Car]. Since the computation layer is not finished[4] and only test nets are active, this cryptocurrency will not be discussed further in this thesis.

2.4 Summary

This chapter discussed the basics of blockchain technologies. First, the basic concepts of transactions as well as the blockchain technology with its different consensus protocols were briefly discussed. Furthermore, the concepts of rewards, forks as well as security issues were covered before a closer introduction to smart contracts was made. There was a short tour of Bitcoin Script, followed by a closer look at the Ethereum platform. This chapter concluded by discussing several top-ranked cryptocurrencies. Thereby, three cryptocurrencies (Ethereum, Tron, Eos) provide support for a Turing-complete language (taking fees out of assessment) in a productive network. While the next chapter introduces state of the art concepts regarding cross-blockchain interoperability, the discussion of cryptocurrencies continues in Chapter 4, where fundamental design decisions need to be made, including the selection of a technology for a prototype.

[4] `https://u.today/cardano-to-launch-new-roadmap-after-iohk-summit`, accessed 2019-03-23

CHAPTER 3

State of the Art

This chapter discusses the state of the art in related work. It first focuses on atomic swap transactions by highlighting different concepts and discussing their benefits and drawbacks. The following section deals with off-chain payment networks that use many atomic swap technology concepts. Then the fundamentals of cross-blockchain tokens are summarized and compared. Subsequently, Section 3.4 discusses further concepts of blockchain interoperability, and Section 3.5 concludes this chapter with a brief summary of the introduced concepts.

3.1 Atomic Cross-Chain Swaps

Herlihy [Her18] describes an atomic cross chain-swap as *"a distributed coordination task where multiple parties exchange assets across multiple blockchains"*. Thereby, the parties do not have to trust each other as the protocol guarantees that the atomic swap only occurs when everyone agrees to the swap and that no one loses any coins, if no agreement is found [Her18].

3.1.1 Implementation

This is usually achieved by using hashlock and timelock constraints. A hashlock h restricts spending an output until a secret s is provided, such that $H(s) = h$, where H is a hash function, while a timelock t restricts the spending of coins until a certain time (block height) is reached. As part of atomic swaps, the locks are combined into a so-called hashed time-lock contract (HTLC). This contract ensures that it will only be executed if the secret s is provided before time t has elapsed. Otherwise the contract is reverting the behavior (i.e., the coins will be reimbursed to the original owner) [Her18]. A graphical representation of the communication on a timeline is shown in Figure 3.1. The explanation is given with the following example.

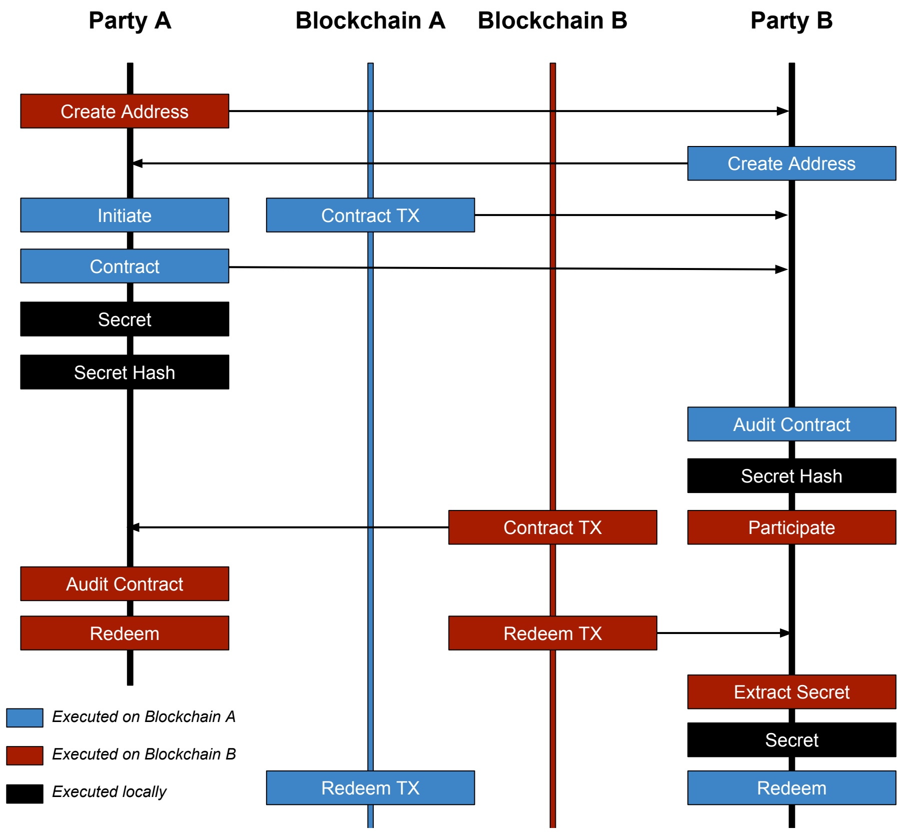


Figure 3.1: Atomic Swap Protocol [ato17]

Example 3.1 *Assume the following atomic swap: Alice (Party A) wants to exchange 1 Bitcoin (Blockchain A) for 50 Litecoins (Blockchain B) with Bob (Party B). The protocol would be as follows:*

1. *Alice and Bob each need an address on the other blockchain (create address).*
2. *Alice creates a secret s, such that $h = H(S)$ and publishes the contract on the Bitcoin platform with a timelock of $\Delta 12$. She sends the contract to Bob.*
3. *Bob confirms the published contract and publishes a contract on the Litecoin blockchain with a timelock of $\Delta 6$ with the same hashlock h found in Alice' contract and sends his contract to Alice.*
4. *Alice checks if Bob has published his contract and reveals the secret to Bob while acquiring Bob's Litecoins.*
5. *Bob extracts the secret from his contract and acquires Alice's Bitcoins by using the secret.*

3.1.2 Security concerns

During the protocol process, there exist many side effects, which may result in the loss of coins. Possible issues include: not forwarding the secret, the order of contracts, and short timelock values. The following list summarizes these issues [Her18]:

- Refunds are triggered by preventing the disclosure of the secrets to others. Unless the first person is not disclosing, the person wishing to boycott the swap loses its assets and will not do so.
- If Bob is deploying his contract before Alice, she does not have to deploy her and can claim the coins because she has chosen the secret.
- When Bob's timelock difference to Alice is almost zero, Alice can reveal her secret at the last second, leaving Bob no time to claim his coins.
- If Alice publishes her contract and reveals the secret before Bob deploys his contract, Bob can claim the coins from Alice without publishing his contract.

3.1.3 Discussion

There exist several slightly different implementations, most of them based on the same concepts [ato17, bar17, bita]. The only requirements for an atomic swap are the support for the above mentioned operations (hashlock and timelock) and the use of a hash function supported by both blockchain technologies. Since transactions are executed in a linearly order, someone has to send the coins first. Therefore, although the correct procedure is followed, a delay or denial of service in one blockchain may result in the loss of the coins of one participating party if the other is dishonest. Thus there is a minimal risk for the participating parties to lose their coins.

There are several approaches for addressing these issues. One of these is BarterDex [bar17], where two UTXOs are required per party. It has additional motivational concepts, for example, Bob has to provide a total liquidity of 215%, with 115% being used as deposit to prove willingness and Alice has to pay a fee that can be lost if the transaction is discontinued. In addition, reputation rankings are calculated on the BarterDex exchange on the basis of successful swaps. These concepts should increase the commitment of the users to complete a started atomic swap [kom18]. Since it merely adds additional concepts for penalizing misbehavior, this thesis does not elaborate on the exact implementation details of this protocol.

3.2 Off-Chain Payment Networks

Off-chain payment networks were mainly discussed with the introduction of the Lightning Network [PD16] in Bitcoin, which was implemented to improve the scalability of Bitcoin by using a network for micropayment transactions. The basic idea is that the transfer of money takes place off-chain and only the final state is published to Bitcoin. Therefore, fewer fees have to be paid and fewer transactions have to be processed by the network. However, this poses challenges for establishing a secure off-chain bi-directional communication channel between two participating parties as well as for finding payment networks which can be combined to further reduce the on-chain transactions [PD16]. There exist also approaches for other blockchain technologies like the Raiden Network [raib] for Ethereum. This section discusses the technologies used by the two highest-rated cryptocurrencies, the Lightning Network as well as the Raiden Network.

3.2.1 Lightning Network

The Lightning Network protocol consists of three phases, a funding transaction, temporary balance updates by commitment transactions and a settlement transaction.

Funding Transaction. For initializing an off-chain transaction, a shared payment channel has to be created. Therefore, the two participating parties create an on-chain multi-signature transaction containing the amount of coins each party has to deposit into the channel. However, before the funding transaction is published, the first commitment transactions are signed and exchanged between the parties for security reasons [PD16].

Commitment Transaction. For each payment in the channel, two commitment transaction are generated, one for each party, which are signed by both parties. The commitment contains the same UTXO and thus only one commitment can be spent on the blockchain. The difference is a restriction for the owner of the commitment. A commitment consists of the following rules:

- The owner can spend his coins after waiting for a certain number of blocks (timelock), or the coins can be spent by the other party by providing a secret whose hash is equal to a value (hashlock constraint). The value is selected by the owner.
- The other party can spend its coins immediately after the publication of the commitment.

If a new commitment is created, the owner reveals the old secret to the other party. This prevents malicious behavior by publishing an old commitment, since the other party can first claim the coins due to the timelock constraint of the commitment owner and the revealed secret of the hash [PD16].

Settlement Transaction. If both parties decide together to close the channel, they can create and publish a so-called settlement transaction that contains the signatures of both parties. Sometimes, however, only one party wants to close a channel. Therefore, the last valid commitment transaction can be used that is published on the Bitcoin network. Since the other party does not know the latest secret (it must be unique for each commitment), the other party can only claim its own coins. Indeed, the owner of the commitment can only spend its coins after the timelock has been reached [PD16].

Payment routing. Since opening a channel between each party is costly (coin deposit is required), a coin can be sent to a third participating party by using a multi-hop payment network. One way to ensure that coins are forwarded in the network can be achieved by using HTLC. The detailed concept is beyond the scope of this thesis and will not be further discussed [PD16].

Example 3.2 *Suppose Alice wants to send coins to Carole. She only has a network with Bob and Bob has already established a network with Carole. Alice can send the coins to Bob, who forwards the coins to Carole.*

3.2.2 Raiden Network

The Raiden Network works in a similar way as the Lightning Network, but is less complex because of the support of smart contracts. First, the participating parties send their tokens to a smart contract where they are stored until the payment channel is closed. While participants in Lightning are using commitment transactions, participants in Raiden are issuing balance proofs signed by the sender to each other, which contain the total sum of all balance proofs sent to the other party. As the amount continues to rise, the party has no incentive to present an older proof at the end. Therefore, the recipient only retains the last proof. These proofs are used for closing the channel. This is done by one party by calling the close function with the balance proof received from the other party. The second party submits its received balance proof by calling the *updateTransfer* function to receive its tokens, or waits for the timeout if there is no balance proof. Subsequently, anyone can call a settle operation to withdraw the money to both participants [raib]. Furthermore, the network supports partial withdrawals and on-chain updates without closing the channel [raia]. However, these are just additional contract features and not of further interest in this thesis.

3.2.3 Discussion

While Bitcoin offers a more complex solution with technologies that are mainly known from the atomic swap protocol, Ethereum has the advantage of supporting smart contracts, facilitating the exchange of commitment messages and supporting additional features that are difficult or even impossible to implement in Bitcoin. However, the number of transactions in Bitcoins is lower by one, since the *updateTransfer* function is not needed. This is only relevant when there is a high fee.

3.3 Cross-Blockchain Tokens

While atomic swaps provide the possibility to exchange tokens between parties on multiple blockchains, cross-blockchain tokens focus on using the same token on multiple blockchains. Therefore, a concept for transferring a token between parties on different blockchains is required, e.g., there must be a technology for sending a token from the Ethereum blockchain to a user who has an account on the Eos blockchain. At the moment and to the best of our knowledge, there exist two protocols for supporting cross-blockchain tokens: Metronome [Aut18] and Deterministic Cross-Blockchain Token Transfers (DeXTT) from the TAST [BRMS18b] project. The following sections are introducing the concepts of these approaches and discuss the differences between them.

3.3.1 Metronome

The Metronome token MET is currently operating on the Ethereum blockchain. The first successful chainhop in a testnet between Ethereum and Ethereum classic was announced at the end of the first quarter 2019[1]. Therefore, a live version is not yet deployed and the following description may be incomplete or may have changed after the release of the first version. Additionally, this thesis focuses on the technical part published by Metronome and does not address the economic aspects contained in its publications such as coin generation and inflation rate.

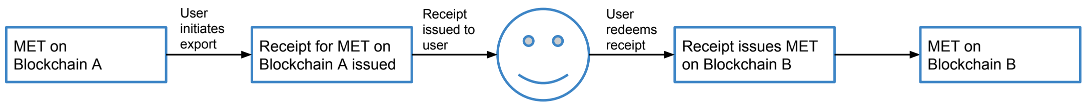


Figure 3.2: Metronome Cross-Chain Token Exchange [Aut18]

Metronome allows user to move their tokens between different blockchains. Therefore, the user has to remove the tokens from the source blockchain, gaining a "proof of exit" Merkle tree receipt, which has to be provided to the Metronome contract on the target blockchain to receive the tokens back. For an overview of this process, see Figure 3.2. This concept is called the "Import/Export System" [Aut18].

Export. When the user is sending coins to a different blockchain, the coins are exported. Thereby, an export function is called, which burns the coins on the source blockchain and issues a receipt to the user [Aut18]. The receipt contains parameters of the export and a hash of a Merkle root tree, which is calculated based on the hash of the previous burn transactions on the source chain. The source chain itself stores all details about the burn [Aut19].

[1] `https://medium.com/@MetronomeToken/on-testnets-chainhops-validators-and-whats-next-57af7d6fd875`, accessed 2019-03-24

Import. The user calls the Metronome contract on the target chain with the receipt received at the export step. This receipt contains the root of a Merkle tree, which is stored by the importer. After the validators have provided sufficient information to verify the validity of the receipt, the contract issues the tokens to the user [Aut18, Aut19].

Validation. Since multiple distributed blockchains are used, Metronome uses a validation concept to verify the import and export steps. So-called validators observe the blockchain by listening for events, validating cross-chain transactions, and voting on the validity of an event. The release of the validators is grouped into three phases. In the end, the validators build up an own cryptocurrency network, where each validator (node) contains a full list of cross-chain transfers. For reaching consensus, a version similar to PoS is used. Furthermore, validators have to deal with issues not encountered with traditional blockchain technologies, like hard forks, where a chain has to be selected, which contains the tokens. Validators receive a fee for validating the receipts, which is paid by the exporter and is distributed equally to the validators who vote for the receipt. A summary, how the import, export and validate step work together is shown in Figure 3.3 [Aut19].

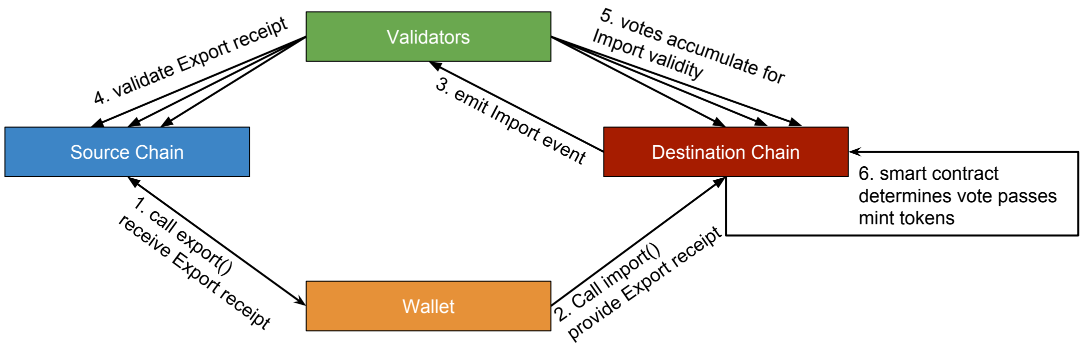

Validators
4. validate Export receipt
5. votes accumulate for Import validity
3. emit Import event
Source Chain
Destination Chain
6. smart contract determines vote passes mint tokens
1. call export() receive Export receipt
Wallet
2. Call import() provide Export receipt

Figure 3.3: Cross-Chain Token Transfer Overview [Aut19]

Voting system. A validator can vote either positive (+1) or negative (-1) for a receipt. Therefore, the validator generates and signs the Merkle path provided for verification from the last 16 burn transactions performed at the source chain. The importing contract verifies the signature of the validator by knowing the public key of the validator and compares the root of the provided path with the root provided by the wallet during the importing step. This procedure is shown in Figure 3.4 [Aut19].

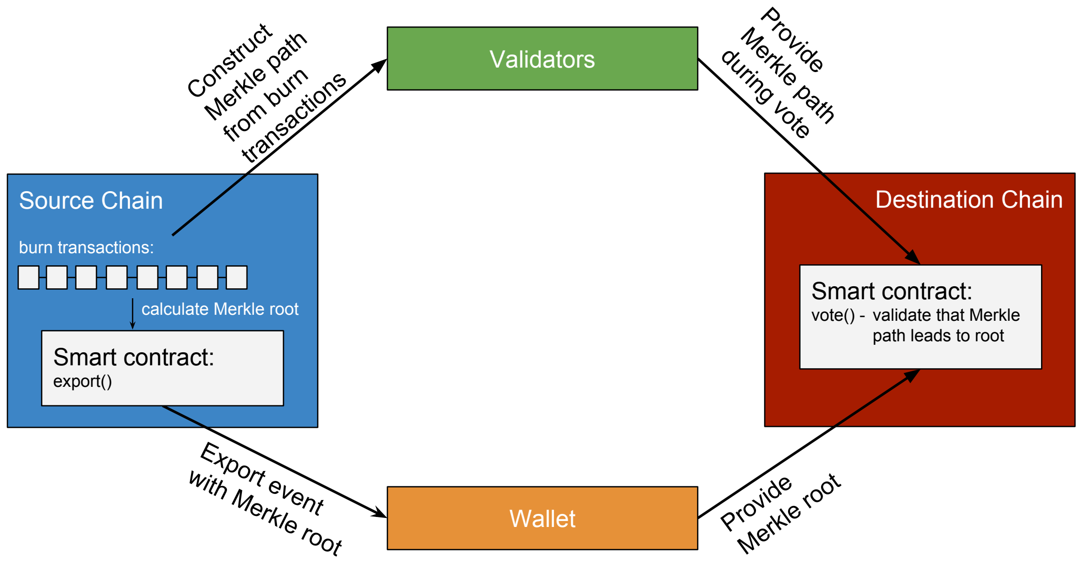

Validators
Construct Merkle path from burn transactions
Provide Merkle path during vote
Source Chain
burn transactions:
calculate Merkle root
Smart contract:
export()
Destination Chain
Smart contract:
vote() - validate that Merkle path leads to root
Export event with Merkle root
Wallet
Provide Merkle root

Figure 3.4: Metronome Import Voting System [Aut19]

3.3.2 Deterministic Cross-Blockchain Token Transfers (DeXTT)

Like Metronome, the first version of DeXTT was written for smart contracts for Ethereum and is not yet used in a productive environment. Instead of transferring the tokens between chains like Metronome, this approach synchronizes the amount of tokens between all chains. This is achieved by witnesses, which verify and broadcast the transactions and by so-called claim-first transactions [dex19]. The following paragraphs describe the protocol in more detail.

Proof of Intent (PoI). For sending x tokens between a source S and a destination D, the two parties have to sign a so-called Proof of Intent (PoI). Thereby, S defines a validity period $t_0..t_1$ of the transaction and signs the information with its signature sig_s, which is countersigned by D with its signature sig_d. This PoI is denoted as $[S, D, x, t0..t1, sig_s, sig_d]$. The created PoI can be used by any participant to verify that this token transfer is intended by both parties [dex19].

Claim First Transactions. While the PoI can be generated on-chain as well as off-chain, it must be distributed across all chains so that each chain can update the tokens accordingly. The next step is the announcement of the PoI, where D publishes a so-called claim transaction on any participating blockchain so that possible witnesses are informed by observing the blockchain (i.e., using events) and forwarding the transaction to all other blockchains [dex19].

Witness Contest. Since witnesses are propagating the information between multiple blockchains, a so-called witness contest is determining which witness will receive a reward fee. Therefore, a so-called contest transaction is generated in which the PoI is signed by a witness with its signature sig_w. This transaction is then published on every blockchain. Thereby, it is assumed that each witness is posting the transaction to each blockchain, maintaining a consistent token balance between the chains when winning the contest, which has the positive side effect that the transaction is distributed across all participating blockchains [dex19].

Finalizing Token Transfer. The contest is running as long as t1 is not reached. Then, the winner is selected by choosing the witness with the lowest sig_w. The decision is calculated by calling a finalize function, which is usually called by the receiver of the transaction (D), but can also be called by any other party. This function transfers the witness reward r to the winning witness and the tokens $x - r$ to the destination D. Since a witness has posted the winning transaction to all blockchains, the lowest sig_w can be calculated by all blockchains in a deterministic way. This would not have been possible if a method has been selected as winner strategy where the order of the winning witness is depending on the transaction order, like choosing the first witness that publishes a verifying transaction [dex19].

Example 3.3 *Assume Alice (A) has a balance of 50 tokens and wants to send 10 tokens to Bob (B) in the next minute. The witness reward is defined as 1 token. There are two witnesses Charlie (C) and Dave (D). The protocol is executed as follows:*

1. *Alice signs a PoI of the form* $[A, B, 10, 1..61]$. *The resulted signature is 0xAA. Alice sends the PoI to Bob.*
2. *Bob countersigns the PoI of the form* $[A, B, 10, 1..61, 0xAA]$. *The signature of Bob is 0xBB.*
3. *Bob publishes the PoI* $[A, B, 10, 1..61, 0xAA, 0xBB]$ *on a blockchain.*
4. *The observers Charlie and Dave are observing the transaction. Each one independently calculates a contest transaction by signing the PoI. It is assumed that the witness signature of Charlie is 0xCC and of Dave is 0xDD. Both publish the contest transaction during the period on the other blockchains.*
5. *The transaction time is over. Bob calls the finalize method. Since Charlie's signature is lower (0xCC < 0xDD), Charlie wins the contest. Bob gets 9 tokens and Charlie gets 1 token.*

Prevent Double Spending. Since the coins are deduced at the end of the period, the sender S can sign two transactions during one time period. If evaluated without verification, the balance of S would be negative after calling the finalize transaction.

Therefore, a veto contest has been introduced, which follows the same technique used for rewards at the contest transaction. Thereby, the witness generates a veto transaction, which contains the conflicting data of both transactions and signs it. Once a veto transaction was recorded, all valid PoIs mentioned in the veto are aborted, the balance of the sender is set to zero and the veto contest is started. The veto contest ends after both PoIs have reached the end of their period plus an offset of the longer time period has passed additionally. Then, the finalize veto function can be called, which pays the reward to the winning witness [dex19].

3.3.3 Discussion

While both approaches set on a technology of using trusted "validators", the concept of transferring and verifying tokens is very different. Metronome "destroys" the tokens to transfer them to a different chain and uses Merkle trees to verify a successful export, whereas DeXTT syncs the tokens across different blockchains through witness contests. Both approaches have their advantages and disadvantages.

Starting with Metronome, exporting tokens by destroying them on one chain and importing them on a another is actually not the same token. It is just a currency that has the same value regardless of the blockchain. Sending money directly to a party on a different blockchain is not possible with the description currently available, since the receipt has to be added by an intermediary on the other blockchain and this concept is not mentioned in any document so far. In addition, the actual draft of the voting script only shows that the wallet has to provide the Merkle root, no additional information is provided there, which can be traced back to bad documentation, since the API documentation of the owners manual [Aut18] reveals some fields for the exporting step, like a *destinationRecipientAddr*. Hence, it can be concluded that the import address must be specified in the export. However, the rewards for the verification are equally balanced between the validators and not tied to any time constraints.

In contrast, DeXTT has no restrictions on sending transactions between parties. For that, a concept called PoI is used, which proves the willingness of both parties to participate in the token transfer. However, a malicious witness can participate only in a few blockchains, which results in inconsistent token balances between the chains. In addition, this can happen when a witness can only send the transaction to a limited number of chains in time. Furthermore, honest witnesses have to download all the chains to participate in the network, which requires a large amount of data storage. Moreover, the cost of the verification messages and finalize messages are quite high and may prevent witnesses to participate in the network, since only the winner is receiving a reward.

In summary, the transaction concept of DeXTT is more mature due to real cross-chain tokens, while the verification concept of Metronome is more promising because it looks more consistent and has fewer constraints to function properly.

3.4 Further Concepts for Blockchain Interoperability

This section summarizes some further concepts of blockchain interoperability.

- **Blockchain Migration**: Based on various factors such as performance, cost and time, a migration to another blockchain is considered. There are several ways to transfer the data, ranging from a fresh start via the transfer of a state only to the complete blockchain history including the block headers. Bandara et al. [BXW19] discuss recurring design patterns for blockchain migration and Lu et al. [LXL+19] introduce a unified blockchain as a service platform that *facilitates the design and deployment of blockchain-based applications.* Since the technological scope of such scenarios differs from cross-platform interactions, this topic is not further discussed.

- **Polkadot [Woo17]**: Polkadot uses a relay chain to distribute transactions to multiple "worker chains" called parachains, which process the transactions in parallel. It includes external blockchains like Ethereum via bridges. At the moment, only a high-level description of Polkadot's functionalities is provided. Thereby, validators forward transactions to Ethereum, and events and Merkle proofs back to Polkadot. The validity of a block is proven by two-thirds majority and by challenges that can be submitted after the vote in a certain time period to prove the invalidity of the header data of a block. Polkadot wants to support not just tokens but all kinds of assets.

- **Aion [SN17]**: Aion introduces a blockchain that connects via bridges to other networks, such as Ethereum. The description is very vague and high level, but the transactions are imported and validated as already discussed in other concepts, e.g., Metronome or Polkadot. A transaction is confirmed when two-thirds of the validators have published voting information and fees are used as incentive.

- **Cosmos [Ten19]**: Cosmos uses a blockchain called hub, which is connected to independent blockchains, called zones, to transfer tokens as packets between the zones through a inter-blockchain communication protocol. The receiving chain has to prove a packet by keeping track of the block headers of the sender chain. Cosmos focuses on token-transfers and has already launched the cosmos hub[2].

- **Ion Stage 2 [Cle18]**: Ion explains how to continue a transaction on one blockchain that depends on a particular state of another chain. For this purpose, a state event is emitted, which is submitted with the corresponding block to the other blockchain. There, the block is verified and the transaction is executed with the state that was transferred by the event. An information who executes the import and who validates the validity of the block is not given. This concept does not focus on cross-blockchain interactions, but only discusses the transfer of states between blockchains and will therefore not be discussed further.

[2]`https://hub.cosmos.network/`, accessed 2019-06-12

3.5 Summary

This chapter introduced the concept of hashlock and timelock constraints and how they can be applied in a practical environments like atomic swaps or off-chain payment networks. Thereby, possible protocols were discussed. However, all protocols have a certain time limit in order to avoid losing coins. This can be used as an attack vector by blocking the submission of transactions by other participants with denial of service attacks. The main part of this chapter was focusing on cross-chain token swaps by taking a closer look at the implementations of Metronome and DeXTT. Both approaches use a kind of intermediary, which is distributing and validating the token transfers. However, both implementations use completely different concepts for achieving this step. While Metronome uses Merkle tree proofs and votes, DeXTT uses signatures and veto transactions. Finally, a brief overview of other blockchain interoperability technologies was given. There, blockchain migration, financial token transfers (Cosmos) and initial ideas to interact with other blockchains were presented (Polkadot, Aion, Ion). However, most of the proposed technologies are described only on a high level and focus only on token transactions and do not take account of smart contracts that have specific characteristics, which will be discussed in Chapter 4. The concepts presented in this chapter will be used as basis in the following design phase.

CHAPTER 4

Design

This chapter presents the design for invoking smart contracts across blockchains. Therefore, this chapter first selects a smart contract platform based on the assessment in Chapter 2. Next, use cases that are helpful for the design procedure are discussed. Subsequently, possible design options are introduced in Section 4.3 followed by an architecture description in Section 4.4. This chapter finishes in Section 4.5 with a discussion of the design.

We define for this chapter the following terminology to distinguish between the elements in the design:

- **Caller.** This describes the account (user/smart contract) that wants to execute a function of an account on a different blockchain.
- **Callee.** This describes the account which will be called by the caller.
- **Source Chain.** This blockchain contains the account of the caller.
- **Target Chain.** This blockchain contains the account of the callee.
- **Distribution Contract.** This contract contains the logic for handling cross-blockchain calls on the source chain.
- **Invocation Contract.** This contract contains the logic for handling cross-blockchain calls on the target chain.
- **Intermediaries.** These users are playing the role of a broker and are transferring the information between the source and target chain.
- **Validators.** These users are validating the information passed by the intermediaries. For simplicity, this group consists of the same users as the intermediaries, however the users are called validators to distinguish between the tasks.

4.1 Blockchain Selection

As discussed in Chapter 2, there are only three possible candidates currently offering comprehensive support for smart contracts: Ethereum, Tron and Eos. Since Tron is a fork of Ethereum and supports the same features, its usage would not provide any benefit over Ethereum, and therefore, we do not consider it as a true alternative in this thesis. Therefore, a decision has to be made between Eos and Ethereum. While Ethereum has a long history in the market, Eos is still in its infancy. Although Eos offers more possibilities and no transaction fees, Ethereum is more widely known, offers mature building tools and is thorougly discussed in the literature [Tik18, ABC17]. Therefore, we select Ethereum as the technology for the prototype. Because the development of smart contracts is expensive, a local blockchain environment is used that simulates the Ethereum core infrastructure. The setup and the tools used are described in Chapter 5.

4.2 Use Cases

Although the main purpose of this thesis is to find a way to interact with smart contracts across different blockchains, there are some use cases where calling a smart contract is not enough, but just the basis. Therefore, this section outlines possible scenarios that may be of interest for smart contract developers when invoking smart contracts across blockchains. These scenarios are derived by an extension of the motivational scenario of Section 1.1, which is given at the end of each list item.

1. **Restrict the function access to specific users.**
   In many smart contracts, for example lottery applications, only the owner or a group of users is allowed to call additional functions. If a user only exists on one blockchain and has no address on the target chain, there must be another mechanism for deciding whether the function of the smart contract can be executed.

   Extension:
   "CrazyParts" offers a special function for "CarTech" where they can automatically file a complaint if "CrazyParts" has not delivered its materials on time.

2. **Sending and receiving tokens.**
   Although cross-chain tokens exist (as described in Section 3.3), there is a possibility that a user might want to use the concept of smart contracts to transfer tokens.

   Extension:
   "CrazyParts" wants to automatically review the complaint and pay "CarTech" a penalty if the complaint was correct.

3. **Chain of smart contract calls.**
   As with smart contract calls for one blockchain, smart contracts should be able to be called recursively between the blockchains as long as there is sufficient gas available.

   Extension:
   A car manufacturer named “HighTechCar” uses a different blockchain than “CarTech” and wants to use the statistical calculation of “CarTech” in its own calculations to predict the delivery dates.

4. **Continue with the execution after invoking a smart contract.**
   The result of a cross-blockchain is available only after some time, since the block creation time varies. The smart contract should continue when the result is returned to the smart contract.

   There are two different ways to continue a smart contract: synchronously and asynchronously. Either the transactions are halted until the result is obtained to proceed with the logic (synchronous) or the transactions continue without waiting for the result (asynchronous). Since it depends on the use case of the caller which option is preferable, both variants should be considered.

   Extension:
   “CarTech” would like to start its analysis process automatically after receiving the result, so that no manual invocation of the contract is required to start the analysis (synchronous).
   “CarTech” is not interested in an immediate result while submitting complaints, since it will receive tokens if the complaint is successful (asynchronous).

4.3 Method Design

4.3.1 Design Preliminaries

Based on the literature, e.g., [Aut19, dex19], the following design for invoking smart contracts across blockchains will also use the concept of intermediaries. This is necessary, since it has to be assumed that the initiator does not have an account on the target blockchain. In contrast to DeXTT, a smart contract is only executed on one blockchain. Therefore, calling a smart contract is more akin to the Metronome project, where a transaction is exported and imported into the target chain. Accordingly, a smart contract invocation always consists of two separate transactions in two directions: calling a method with parameters and returning the values to the initiator.

The handling of cross-chain smart contract calls is more complex compared to the handling of cross-chain token transfers. While in cross-chain token transfers, the gas cost for submitting the transaction is known in advance (always the same code, therefore the same gas required), the amount of gas for executing a cross-chain smart contract is individual and cannot be determined beforehand in all cases (e.g., unknown variable sizes).

Thus, it cannot be guaranteed that the gas paid by the caller is sufficient for executing the cross-chain smart contract transaction. Therefore, the caller has to ensure that the executing intermediary receives not only a static fee but also the gas cost. Otherwise, the incentive to execute the transaction decreases. Unlike DeXTT, where the winning witness is selected at the end of the token import, the intermediary has to be chosen in advance, as a caller does not want to reimburse the gas cost to multiple intermediaries as the cost is increasing linearly and an intermediary does not want to execute the transaction without getting the gas cost refunded. Possible methods for invoking smart contracts are described in Section 4.3.5.

Compared to the complexity of calling methods, it is easier to return values to the source chain. There, only the received values have to be returned to the source chain, which is similar to the import step in Metronome, since the required gas depends only on the length of the return value.

However, if more complex interactions have to be considered as well, e.g., the continuation of a transaction after receiving a return value, the advantage of using a fixed gas value is no longer given. Therefore, the method design has to take this into account, together with all other use cases mentioned in Section 4.2. Possible methods for passing back the return values are described in Section 4.3.6.

There are also general issues that need to be considered, for example, use cases such as restricting method access for specific users, requirements like dealing with consistency issues or protocol tasks such as the selection of participating intermediaries. These requirements are covered in the following sections.

4.3.2 Concurrency & Consistency

When different blockchains are considered as a common ecosystem, we cannot say anything about the order of execution of the transactions across the blockchain boundaries, since the block creation time varies (see Example 4.1), whereas the transactions on a single blockchain are guaranteed to be executed in a sequential order [DGHK17]. This means that while a transaction is processed on one blockchain, a different transaction is processed on a different blockchain. While this does not pose a problem for the following method design, it might create issues for developers using the proposed system. Developers have to consider that a response from a transaction executed later may be obtained earlier than a transaction executed earlier, since the order in which other blockchain systems perform certain transactions cannot be determined. Therefore, a developer has to be aware of the transaction status for cross-blockchain invocations, since the final result may not be available immediately as in a single blockchain, but may arrive later. This behavior is well-known from other technologies like asynchronous communication [ZBZ11].

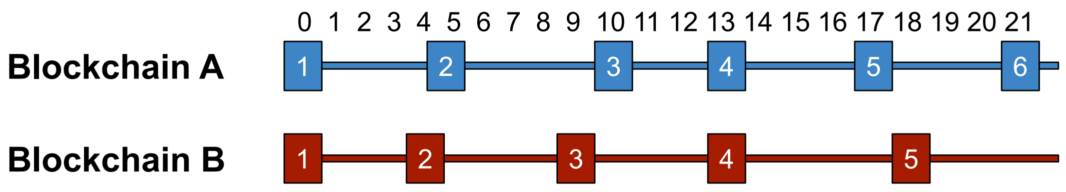


Figure 4.1: Execution Order of Two Blockchains

Example 4.1 *It is assumed that two different blockchains (A and B) have been started at the same time with an average block time of 5 seconds and only one transaction per block. Since PoW is used and a nonce has to be found (see Section 2.1.6), the block-time varies randomly. This visualizes, for example, etherscan.io*[a] *for Ethereum with a range between 13 and 30 seconds. The transaction diagram shown in Figure 4.1 illustrates that it may happen that the transactions are executed in parallel (time 0 and 13) or that a transaction is called before or after the transaction of the other blockchain (A before B: 17-18, 21; A after B: 4-5, 9-10). Thus, nothing can be said about the order of transactions between two blockchains.*

[a]https://etherscan.io/chart/blocktime, accessed 2019-05-24

However, the method design is strongly influenced by the consistency behavior. Since it cannot be guaranteed that the received transactions are valid because of the eventual consistency of blockchains, there is a common approach to wait for a certain amount of blocks before the transaction is considered as valid. This approach is not only used by exchanges [Com18], but also proposed by Metronome [Aut18]. Finding a reliable number of blocks requires a tradeoff between security and time. While Metronome generously sets the number to 4000-5000 blocks (one block per 15 seconds in Ethereum[1] means a waiting time of about 17 hours), exchanges use smaller confirmation numbers. For example, Binance, a large exchange service, uses 30 confirmations[2] (about 7.5 minutes). Metronome argues that this high number avoids forking and mitigates the effects of a 51% attack [Aut19]. Since exchanges are also affected by a 51% attack, this argumentation is not fully comprehensible.

An approximate calculation that compares the numbers of Binance and Metronome (Metronome in parentheses) shows that a transaction is executed after 7.5 minutes (17 hours) at the target chain and the return value is received after 15 minutes (34 hours) and considered as safe after 22.5 minutes (51 hours), if no additional steps are needed. As the numbers show, a longer waiting time than necessary postpones the function call and, considering the usage scenario of the car manufacturer, leads to a delayed comparison of the values and thus to a later intervention in the production chain.

[1]https://etherscan.io/chart/blocktime, accessed 2019-03-29
[2]https://www.binance.com/userCenter/deposit.html, accessed 2019-05-07, login required

4.3.3 Selecting Intermediaries and Validators

For transferring the data between the source and the target chain, so-called intermediaries are used. These intend to transfer the data of the cross-blockchain calls between the source and target chain. Since an intermediary can act dishonestly, so-called validators review the actions of the intermediaries and inform the distribution contract that something has been suspicious.

In order to ensure that mostly honest participants act as intermediaries and validators, they have to be selected. Metronome relies at the final stage on the approach of setting up its own blockchain to manage and select their validators, which can be applied to our work as well. However, in our work, the intermediaries and validators are created on-demand, as it is just a proof-of-concept, which does not require a detailed implementation of a selection procedure for testing the functionality. Therefore, we do not focus on the selection of possible intermediaries and validators in our work.

4.3.4 Restricting Access to a User

Restricting the access of a specific user is usually implemented by checking whether a sender address of a message exists in a stored list [Etha]. However, this is not possible in cross-blockchain interactions, since the sender of the message is not the user but the intermediary. Therefore, it is necessary to pass the initiator of the transaction in a different way so that the authorization is granted for executing the method on behalf of the user. Furthermore, there are some requirements that have to be considered:

- The intermediary is not known in advance. Therefore, the approval cannot be tied to a particular intermediary without introducing additional transactions that increase the delay and complexity of the protocol.
- The permission should only be granted for one transaction and be bound to the parameters of the users. This limits the possibility to execute a different function on behalf of the user. For example, a dishonest intermediary cannot call a destroy function.

There exist well-known concepts that can be used to solve the issue. One of them is the use of digital signatures [RSA78]. For this, both blockchains have to support the same asymmetric encryption and hash function. At first, the caller concatenates all parameters, then calculates the hash value and signs it. This hash value is then additionally passed to the smart contract, which can verify whether the parameters are chosen by a valid user.

To prevent replay attacks, the number of executions can be limited by using a random nonce [LKKY05]. The contract keeps track of used nonces and blocks execution if a nonce is reused. If a different party executes the call with the signed parameters before the intermediary, then this call is still executed at the discretion of the user. Even if the transaction runs out of gas, this is not an issue (except the loss of the gas) as the status is reverted, the nonce is not added to the list and can be re-executed with sufficient gas. Therefore, a different party cannot disrupt the execution.

However, saving all nonces requires storage space and increases the complexity and cost of invoking smart contracts. Therefore, this option should only be used on purpose, e.g., when an approval of an user is required.

4.3.5 Invoking Smart Contracts

This section discusses the steps for transferring the transaction from the source chain to the target chain. First, we perform an analysis to define the phases required for executing a cross-chain transaction, which are discussed in detail afterwards.

Sequence Analysis

For transferring a transaction to the target chain, we rely on intermediaries as discussed in Section 4.3.1. In order for an intermediary to become aware of a call, it has to be announced by the caller. This can be done either by finding an intermediary off-chain or by using an on-chain transaction. Since finding an intermediary off-chain requires an additional infrastructure already provided by the blockchain, an on-chain transaction is used.

As discussed in Section 4.3.1, only one intermediary should perform the cross-blockchain call. Therefore, an intermediary has to claim the right to execute the transaction. Possible options are the use of the first intermediary claiming the right, the selection of the caller, or the selection by an algorithm, such as using the lowest signature as defined by DeXTT. The first two options cannot be used because the use of the first intermediary prefers intermediaries with a better internet connection or the miners who mine the block, and the selection of a caller requires additional interactions by the caller, which should be avoided because a smart contract cannot invoke functions without being triggered. Therefore, we use the last option.

After the intermediary has been selected, it can execute the transaction on the target chain.

Following this analysis, calling smart contracts across blockchains can be divided into the following phases:

1. **Register Phase**: The caller announces the transaction on the source chain.
2. **Offer Phase**: The intermediary makes an offer and commits to execute the transaction.
3. **Executing Phase**: The selected intermediary executes the transaction on the target chain.

Register phase

In the first phase, the caller commits to execute a transaction on a different blockchain. For this, a contract function named `registerCall` is called on the source chain, which passes the blockchain identifier of the target chain, the address, the method and the parameters of the contract as well as the fees paid to the intermediaries and validators to the method. The caller deposits tokens with which the transaction can be executed, i.e., the gas and the fees paid. This function generates a unique invocation id, which is returned to the caller.

Hint 4.1 *In the prototype used in this work, the fees are defined in the contract and do not have to be passed by the caller to reduce the already limited call stack. In a later release, a feature can be added to update these fees by selected delegates.*

The gas price and startgas have to be chosen by one party, and there are various possibilities to choosing this party, as we discuss in the following paragraphs. The selection is dependent on whether additional parameters are required.

- **Caller sets Gas.** If the caller determines the gas price on the source chain, it has to know the current market behavior of the target chain. While the startgas (steps in the contract) remains the same and can be set by the caller (this can be calculated approximately in advance), the gas price is a dynamic market value and cannot be determined by the caller without any additional knowledge. Therefore, selecting the startgas is a valid option, whereas the actual gas price should not be set by the caller. However, if a caller wants to set a maximum limit, it may be an option to include a maximum gasprice field at the request.
- **Intermediary sets Gas.** If the intermediary sets the gas price, there is no control institution that can verify, without discussion, whether the chosen values are acceptable. Thereby, the intermediary can set the value of the gas price very high, so that all gas of the caller is wasted. Furthermore, if the startgas is also set by the intermediary, it can always trigger an "out of gas exception" so that the caller has to invoke the transaction with more gas again. This cycle can be continued arbitrarily to burn the caller's tokens, which is an undesirable behavior and therefore not an option.
- **Target Blockchain sets Gas Price.** There is a contract on the target blockchain that contains a gas price, which must be used by the intermediaries for executing a transaction. In a full infrastructure, where the intermediaries have their own consensus protocol, this value could be adjusted by delegates who were chosen by the users. This has the advantage that the gas price is transparent and fair, since it can be verified by all participants. However, the price is probably higher than desired by the caller, since the caller cannot influence the price. The reason why the source chain does not contain the gas price, although the caller would have the opportunity to see the actual gas price, is mainly because the source blockchain

would have to store the gas price for each blockchain. This leads to updates across all blockchains and thus to many unnecessary transactions.

- **Gas Price set during Offer Phase.** Intermediaries propose a gas price per step in the currency of the caller during the offering phase. This leads to a competition for the lowest offer, as the intermediary with the lowest gas price wins the competition. For this, the respective intermediary itself has to ensure that the ratio between the gas price on the source chain and the gas price on the target chain is not below the current market value, since otherwise the offer is not lucrative for the intermediary. However, if the exchange rate is higher, it will get more gas than the current market value. Therefore, a caller has to set a maximum gas price to avoid excessive offers.

Suggested Solution. At the register phase, the caller sets the maximum number of steps (startgas) as well as the maximum gas price and deposits enough tokens. The final gas price is proposed during the offer phase. The minimum amount of deposited tokens can be calculated as follows:

$$\text{max gasprice} \cdot \text{startgas} + \text{fees} \leq \text{deposited tokens} \tag{4.1}$$

Offer phase

In the offer phase, intermediaries can submit a bid for executing the transaction by calling a `makeOffer` function on the distribution contract. For this, the intermediaries monitor the blockchain for new cross-platform transactions. Once a new transaction occurs, they submit an offer request and thus participate in the contest of being selected as the executor of the transaction.

As has already been mentioned, this request has to include a gas price. At the end of the offering phase, the intermediary with the lowest price will be selected. The formula for calculating the lowest price is as follows:

$$\text{price} = \text{gasprice} \cdot \text{startgas} \tag{4.2}$$

Since each intermediary wants to receive the execution fees, the gas price will normalize to a value similar to the market value. If several intermediaries submit the same price, the intermediary with the lowest signature will be selected. This approach is used in DeXTT and guarantees that the intermediary can be selected in a deterministic way without any additional transaction.

Signature Definition. Since the signature should vary between invocations, the invocation id in combination with the sender address would be a good candidate. However, the incentives to update the price will be low when others have a lower hash since they only have to set the same price to get back into pole position. Therefore, the gas price is included in the hash. This guarantees randomness in the signature calculation and nobody is favored at price updates. Therefore, we define the signature as the hash of concatenating the sender address, the invocation id and the gas price.

The offer phase can end in several ways:

- **Caller-based.** The caller has to send a transaction to complete the offering phase. This has the advantage that the caller can decide whether the offer meets their wishes. However, if a smart contract needs to interact with a different blockchain, this is not an option, since a contract cannot call a function by itself. In addition, the user experience would be degraded if users have to watch the blockchain status and call additional functions.

- **Time-based.** The voting ends after a predefined time. Thereby, the selection is made implicitly with the provided rules after the time has elapsed. Therefore, no explicit call is required. However, the intermediaries have to follow open competitions and cannot receive a notification announcing the end of the phase, so the simplicity of working only with events is no longer given.

- **Random-based.** The phase ends pseudo-randomly after an offer has been submitted by an intermediary, either by using insecure blockchain parameters or external services [Reu18]. Thereby, a notification can be sent to the participants. However, this randomness may lead to higher fees, as the competition may end too soon or may not even end, if the condition is never triggered.

- **Combined Solution.** Defining a function that ends the offer phase by the caller can be added as it works well with the other options. Furthermore, the random-based approach can be combined with the time-based approach so that the phase closes randomly after a fixed period of time. This guarantees that the intermediaries have enough time to submit their offers.

  However, the worst case that an offering phase will never end, remains. One possible solution consists of using a soft and a hard limit. While the offering phase remains active at least until the soft limit, it can randomly close between the soft and the hard limit and will definitely close at the hard limit. Due to the hard limit, the intermediaries still have to follow the competition status (at least while they are in lead) and thus there is no real advantage of such a solution.

  A key factor in choosing the right solution may be the incentives for receiving fair or good offers. It can be discussed, among other things, whether randomness increases the incentive to publish greatly reduced offers or whether reaching a time limit increases voting behavior. Since our prototype shows only the functionality of cross-blockchain smart contract invocations and does not focus on finding the best way to increase the incentive to make fair offers, this research will be kept open for future work.

Suggested Solution. All proposed solutions have some disadvantages, so a trade-off decision has to be made. When no time-based limit is chosen, there is a likelihood that the offering phase will never end. The advantages for using soft and hard limits is not really visibly. Therefore, the proposed solution is time-based only, although it is known that potentially lower fees can be achieved by using other methods, as some intermediaries may not be able to participate in the contest because the competition is over before the transaction has been processed successfully. If no intermediary has submitted an offer, then the cross-chain transaction is aborted. In order to receive the deposited tokens back, the caller has to finalize the call (see Section 4.3.6).

Aborting Transaction. As long as the offering phase is open, the caller can execute an `abort` function that aborts the requested transaction. The intermediary who is currently winning will receive the fee and the remaining tokens will be returned to the caller.

> **Hint 4.2** *The prototype does not implement this feature, since a processed transaction cannot be revoked in a single blockchain either.*

Revoking Offer. An intermediary may want to revoke its offer by calling a `revokeOffer` function before the end of the offer period. This may be necessary if the market changes quickly and the offer is no longer in line with the market. However, this distorts the offering phase since all other offers depend on this value. Therefore, the offering process must theoretically be restarted.

However, there can be a dishonest intermediary who submits a best offer and subsequently revokes it to trigger a restart of the offer phase. If this behavior is repeated again and again for the same transaction, reaching the next phase can be blocked. Therefore, the intermediary has to be excluded for submitting offers for a certain number of blocks. In addition, the intermediary has to pay a penalty fee, which the winning intermediary will additionally receive for the overheads. For the sake of simplicity, the fee is equal to the fee paid by the caller.

> **Hint 4.3** *The prototype does not implement this feature for simplicity. If an intermediary does not want to execute the offer, it can simply avoid executing the next step. The transaction is then flagged as fraud and aborted by the validators.*

Example 4.2 *Assume the following scenario shown in Figure 4.2, which illustrates the discussed elements of the offer phase:*

1. *The caller registers the cross-blockchain call.*
2. *Intermediary A and B submit their offers and try to underbid each other.*
3. *The market value changes. Therefore, intermediary A revokes its offer. This triggers a restart of the offer phase.*
4. *Intermediary B submits a new offer.*
5. *Intermediary A cannot submit a new offer, since it has revoked the last one and is blocked.*
6. *The caller does not like the offer of B since it is too high and aborts the transaction.*

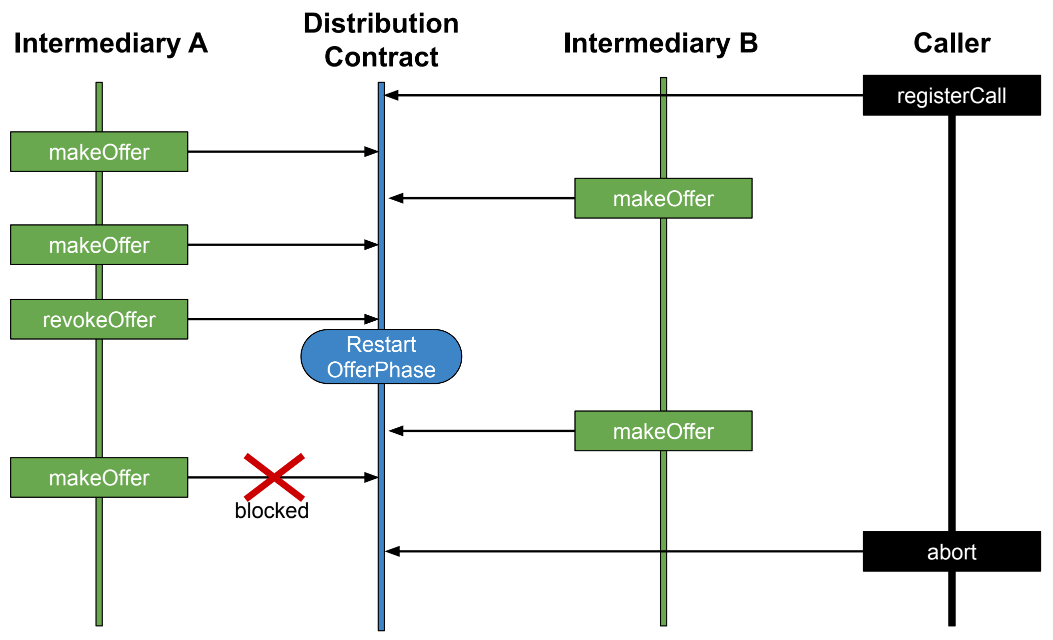


Figure 4.2: Possible Sequence of Offer Phase

Executing Phase

After an intermediary has won the selection, it has to execute the transaction on the target blockchain. Since the contract call with its result can only be checked by the validators, when the contract is called via a smart contract that logs input and return values (the result of the transaction is otherwise invisible to others), the intermediary executes a function named `executeCall` with the data received from the source chain. This function forwards the transaction to the callee and stores the input and return data with a unique execution id.

Penalizing Misbehavior. In some cases, the intermediary gives up its reward and does not execute the transaction. Then, the tokens of the caller would be locked forever. To solve this problem, two options have to be considered: releasing the locked tokens to the caller and making this decision unattractive for the intermediary.

Release Locked Tokens. In order to release the locked tokens to the caller, a method for detecting misbehavior has to be defined. Thereby, two options are proposed: a time-based and a voting-based detection method.

- **Time-based Detection.** The tokens can be refunded after a certain period of time (for example after one week or 40,000 blocks). However, if multiple cross-chain blockchain calls are executed, the time may not be sufficient, resulting in a loss of tokens by the intermediary.
- **Voting-based Detection.** If the caller has doubts that the transaction has been executed, it can create a vote of no confidence by calling a `requestVerification` function. Each validator observes the blockchain whether the intermediary has executed the transaction on the target blockchain by checking the transaction log. Depending on whether the transaction has been found, the validator submits an invalid (no fraud) or a valid (fraud) vote to the source chain contract by calling a function named `fraudVote`. At the end of the voting period, a fee will be paid to the voting winner (where the winner is identified by having the lowest signature and the correct voting result based on the majority of votes). If the voting result is invalid, i.e., the vote of no confidence turns out to be untenable, the transaction continues normally and the fee is paid by the caller (the fee has to be deposited at the beginning of the voting). Otherwise, the fee will be paid by the intermediary, the transaction will be aborted and the locked tokens will be released to the caller.

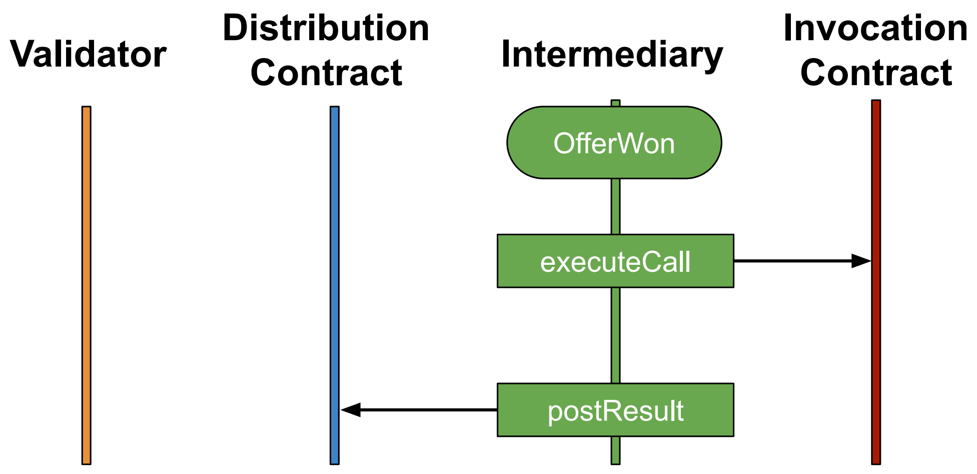
  Under the assumption that at least 51% of the validators behave honestly and the probability of voting decreases exponentially per voting option (if the signature is not lower, the vote will not be submitted because the voter could not win), this procedure should ensure that a honest intermediary does not lose any tokens. In practice, this may not be always the case, since in the worst case the signature at the first vote for a voting option will be zero, resulting in no further votes for this option.

Furthermore, the source chain does not know the truth value of the target blockchain and can only judge by the majority of votes regardless of the correct result. Therefore, validators have no incentive to behave honestly, aim for profit and always vote for the majority if the signature is lower. Thus, it is important to monitor the validators. This can be achieved, for example, with dPoS, where participants will only nominate honest validators.

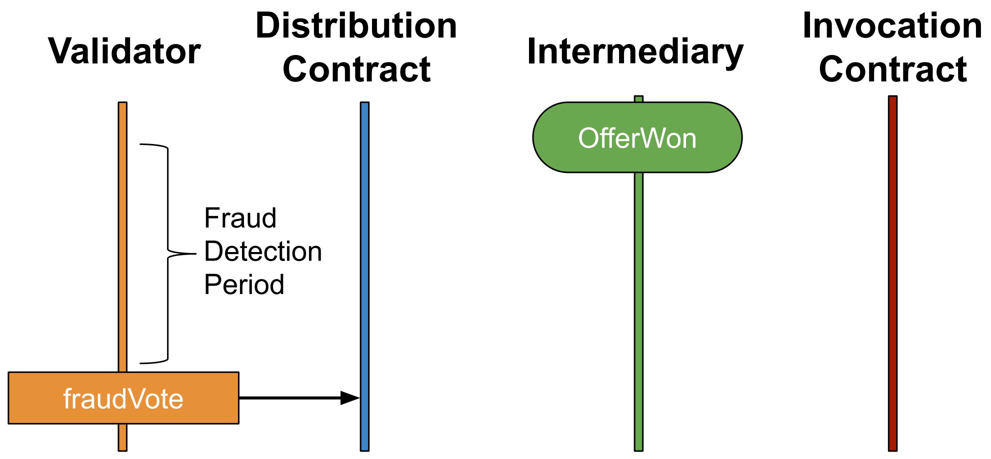
Thus, other options, such as the payment of a fee to all validators or the comparison of the signature with a predefined value (e.g., the block hash that can only by influenced by the miner) may be preferred. A short discussion of voting alternatives is given in Section 4.3.7, although these are not considered further in this thesis.

An extension for the proposed method allows valdiators to vote for suspicion without being prompted by the caller. This reduces the complexity for developers and users of smart contracts because the status does not have to be monitored and no proper functionality has to be provided in the own smart contract for triggering a request verification. The voting period starts with the first vote of a validator. Since the caller cannot pay the fee if the voting is invalid (the number of invalid votings is not known in advance and validators will always trigger this voting to get more tokens), the validator has to deposit the fee for the payment of the other validators, if its voting initiative is wrong. Thus, a validator will only start voting for a valid fraud, otherwise its deposit may be paid to a different validator if its own signature is higher. The rest is defined equally.

Suggested Solution. Since validators have more information about the blockchain and can easily detect misbehavior, only those can start a vote of no confidence. The first voting is allowed only after a certain period of time (e.g., one week), if no result was published. This guarantees to the intermediary that it has enough time to execute the cross-chain invocation and return the result to the transaction on time. If a voting is active while the result is submitted, the voting is aborted and the deposition of the fee is reimbursed to the corresponding voter. Furthermore, a long delay of an action of an intermediary (e.g., publication of the transaction on the target chain after winning is delayed by the defined time period) is considered by all validators as a violation of the rules. This ensures that the voting result is not influenced by a postponed action of the intermediary. Figure 4.3 visualizes the difference between a successful execution and a vote for no confidence. Figure 4.3a shows a valid sequence in which the intermediary executes the transaction (`executeCall`) after winning the offer (`OfferWon`) and submits the result of the call to the distribution contract (`postResult`). In contrast, Figure 4.3b shows that the intermediary does not execute the transaction within a certain period of time after winning the offer. Thus, the validator submits a vote of no confidence (`fraudVote`).



(a) Valid Execution



(b) Fraudulent Execution

Figure 4.3: Difference between a Valid and Fraudulent Execution

Penalty Fees for Intermediary. To penalize the non-execution of a cross-chain smart-contract call, the intermediary has to deposit tokens in a depot, with which the misbehavior can be compensated. For this, an intermediary can deposit any number of tokens at any time (using a function named `depositTokens`), whereby the more tokens are deposited, the more transactions can be executed in parallel. The tokens are locked from the deposit on submission of an offer and get available for further use once the cross-blockchain call has reached the final state or a better offer has been made. Each time a transaction is not executed, the locked tokens are used to pay the caller a compensation fee for the breach of the contract. All unlocked tokens can be withdrawn by the intermediary at any time (`withdrawTokens`).

As part of the development, the following concepts were also developed:

- **Share deposit between all callers** In order to simplify the management of locked tokens, the intermediary has to deposit a static amount of tokens. If the agreement is not fulfilled, all failed transactions registered in an appropriate time period will receive a compensation fee based on a distribution algorithm. The deposition of the intermediary is set to zero. It turned out that this approach increases the complexity in case of misconduct. Additionally, if a withdraw request is executed, it has to be verified that no transaction of the intermediary is in progress.
- **Paying fee while making an offer**: Each time an offer is made, the intermediary deposits tokens which will be refunded after the agreement has been fulfilled. As multiple offers are submitted, any better offer has to reimburse the tokens of the previous offer, resulting in many unnecessary transactions. With a deposit, token management is outsourced to a central location, regardless of a specific requirement.

Because of the disadvantages for handling misconducts in "Share deposit between all callers" and the simplicity of using a centralized token management compared to "Paying fee while making an offer" we do not pursue the approaches any further.

4.3.6 Receiving Data from Smart Contracts

After the transaction has been executed, the return value must be forwarded to the source chain. Again, we first discuss the required steps and then explain the details.

Sequence Analysis

Once a result has been received by the target chain, the intermediary has to forward the result to the caller. This can be done again on-chain or off-chain. Using the same argument as in Section 4.3.5 of an already provided infrastructure by the blockchain, the source chain is used to transfer the result to the caller.

In return for offering the services, the intermediary receives a reward from the caller. If an intermediary receives the reward before executing the transaction, a dishonest intermediary may collect the reward but not proceed with the execution. Therefore, the only way to pay the reward is after the execution.

However, it can be that an intermediary does not behave honestly and forwards the wrong result. Therefore, a verification step is required before the reward is paid and the result is considered valid. Since a caller cannot verify the execution without knowledge of the target chain, the caller needs help to verify the transaction. This is done by using validators, known from the concept of Metronome, who compare the values of the target and source chains. Alternatively, the callee can inform the caller about the successful execution off-chain, but again an external infrastructure would be required.

Therefore, we divide again the section into three phases:

1. **Result Forwarding**: The intermediary forwards the result from the target to the source chain.
2. **Result Verification**: Validators verify the import to prove the correct execution of the intermediary.
3. **Token Claiming**: The fees are paid to the intermediary and the validators.

Result Forwarding

In this phase, the result is forwarded from the target chain to the source chain. Therefore, the intermediary calls a function called `postResult` on the source chain. It passes the result received from the contract or null, if unavailable, the amount of gas used when calling the transaction and the execution as well as the invocation id.

However, the intermediary can be dishonest and may not pass the result back to the source chain. Although the incentive to do so is small (except for token transfers, see Section 4.3.10), since tokens have already been spent for executing the transaction, this may happen. Therefore, the same voting procedure is used as in the execution phase.

Usually, there has to be an unfortunate coincidence, that the value has not been forwarded to the source chain, e.g., the intermediary has lost the internet connection or the transaction has not been processed by the network. Therefore, the vote of no confidence needs to be adjusted by introducing a third state: "result available but not transmitted". Validators pass the correct result to the chain during voting or in an additional step after the voting. As this approach adds complexity to the design, introduces new states, and extends the scope of the prototype, it will not be discussed further. Instead, an undelivered transaction is marked as invalid and the call is aborted.

Side note: A transaction can always be executed with the same parameters more than once, whereby the intermediary may not necessarily be the first one to execute the transaction (another party will execute the call request before the end of the offering phase). Thus, if a unique action is required, the use of nonces as described in Section 4.3.4 and the use of a consistent return value are required. This is achieved by storing the return value in the contract for each nonce. For a concept of token transfers, where the intermediary has to transfer tokens to the source chain, see Section 4.3.11.

Result Verification

The published result has to be verified for correctness. This is done by validators by comparing the following values from the source chain with the values stored on the target chain:

- used gas + max gas
- target chain + contract address + method name

- parameters
- return status (ok, exception) + return value
- invocation and execution id

There exist three possible voting results: call invalid, call valid but used gas invalid, call valid and gas valid. Each validator can vote for a possible result by calling the `vote` function. Like the other votings, it ends after a certain period of time. Then, the positive (call valid) and negative (call invalid) votes are compared. If the vote is positive, the gas votes are compared. The fees will be distributed according to the same criteria described in the previous votings.

Moreover, a brief security discussion has to be conducted to ask if a validator can act dishonestly and trigger a Sybil attack to gain a majority of votes. Since validators are selected by PoA (as defined in this design) or by voting in a dPoS consensus protocol in which the stakes weight the voting rights, it is possible to identify dishonest validators and revoke the voting rights (PoA) or stop voting the validators into the top-k (dPoS). Therefore, Sybil attacks can be excluded in practice as long as the consensus protocols ensure that creating another entity does not increase the voting power.

Token Claiming

After the voting is completed, a `finalize` function is called on the contract of the source chain to distribute the tokens. Depending on the voting result, the tokens are distributed differently.

If the voting result is negative or the gas used is incorrect, the entire transaction value (with the exception of the result verification fee and the gas cost for executing the token claiming step) is reimbursed to the caller. Therefore, the caller or the validators will call the `finalize` function.

If the voting result is positive and the gas price is correct, the intermediary will be reimbursed for its transaction cost plus compensation, the winner of the voting will receive its reward and the remaining tokens will be refunded to the caller. Since the intermediary receives most of the tokens, it is assumed that the transaction will be executed by the intermediary. However, it can also be called by anyone else, such as the caller or the validator.

Upon completion of this phase, the caller can access the returned value by calling the `getValue` function on the contract of the source chain.

If no offer has been submitted, the function has to be called as well to receive the tokens back from the caller as mentioned in Section 4.3.5. Since there is no voting and no execution, neither the intermediary nor the validator will execute the `finalize` function because no reward is paid. Therefore, only the caller will execute this function.

Hint 4.4 *It should be avoided that the caller has to execute the transaction so that the cross-blockchain call can be used by the caller without observing the blockchain. Therefore, a reward may be considered for calling the* `finalize` *function which allows to outsource the call to the validators or intermediaries.*

4.3.7 Voting Alternatives

The previous section describes a solution that limits voting to a certain time period. While this approach is used in the design, it could be improved by additionally waiting for a certain number of votes, so that a minimum amount of validators participated in the vote. For example, in Metronome there must be a minimum number of positive votes to accept the vote. However, there is no possibility that a vote ends negatively, it can only be postponed. This approach is not possible with this design because the transaction value must be paid or refunded depending on the outcome of the voting.

Metronome uses a concept in which all validators receive an even share of the fee. This improves the willingness to vote, but as mentioned earlier, a lot of gas has to be paid to transfer the fees to each validator. As long as the number of validators is limited, the amount of gas used for transferring the fees is within manageable limits. However, if more validators need to be supported, a linear increase in gas cost and a linear decrease of the voting reward per validator has to be expected.

Another approach for improving the willingness to vote, but keeping the gas price low, is to randomly select a voting winner when calling the `finalize` function. However, it is difficult to achieve true randomness because the operations on an Ethereum blockchain have to be deterministic to reach a consensus about the execution state across the network [Ethc]. Thus, only an external factor can be used to make the voting independent of the miners [Reu18]. In addition, if no external factor is used, validators will wait to call the `finalize` function until the chosen factor guarantees its own winning.

4.3.8 Extension: Continuation of Caller Smart Contract

After the result of a cross-blockchain smart contract call has been received, the caller smart contract should be automatically continued. Therefore, the caller has to specify additional parameters (startgas, gasprice and method name of the callback function) at the `registerCall` function that informs the contract that the call should be continued.

As mentioned in Section 4.2, the developer should be able to call the continuation function synchronously or asynchronously. By default. the function is executed asynchronously. However, the developer can simulate a synchronous call by storing the state of the function in storage and ending the function immediately after invoking the `registerCall` function. Then, the function state can be restored when the cross-blockchain result has arrived.

Furthermore, the token claiming step has to be adjusted. When the intermediary executes the `finalize` function, the callback function is executed first and the required gas is truncated afterwards from the provided gas of the caller. The remaining function continues as defined.

Thus, the caller has do deposit more tokens at the beginning. The formula from Section 4.3.5 is adjusted as follow:

$$\begin{gathered}\text{max gasprice} \cdot \text{startgas} + \text{fees} \\ + \textbf{callback startgas} \cdot \textbf{callback gasprice} \leq \text{deposited tokens}\end{gathered} \tag{4.3}$$

The callback gas price is a value that the caller pays per step for executing the callback transaction. Since the gas price is determined by the gas price used for calling the `finalize` function, the callback function will not be called with a higher gas price than defined by the caller. Therefore, only a gas price below the callback gas price is used to benefit from the callback invocation. Since other participants may call this function with a higher gas price (transactions with a higher gas price are ranked ahead), the gas price will be close to the gas price specified by the caller. If it turns out that this competition is not working, checks can be added to confirm that the intermediary has not used a lower gas price.

Hint 4.5 *The prototype implementation does not specify the gas price of the callback function. It assumes that the current gas price of a transaction is equal to the gas price of the* `registerCall` *transaction.*

4.3.9 Extension: Chain of Smart Contract Calls

Depending on the use case, the concept may be extended to forward smart contract calls between several blockchains. If the source chain is not interested in the final result, nothing needs to be adjusted. However, if the source chain needs the final result of the chain, the complexity is increased.

In this case, the invoked contract has to return a predefined status so that the invocation contract at the target chain is aware of this state and notifies the intermediary that the value is not yet intended for the source chain (status pending) and therefore cannot be published. The intermediary has to wait until completion of the cross-chain call. After the result of a transaction is received, the pending transaction has to be updated with the new value. The pending status is replaced by the return status and the return value is added. Then, the transaction is forwarded to the source chain by the intermediary.

When a new cross-chain transaction is invoked while a transaction continues, the pending status is updated with the new cross-chain transaction. Once the new result is received, the process continues as defined.

Figure 4.4 shows an abstraction of such a chain using three methods, where the final method does not trigger a cross-chain transaction. Each color of an arrow shows a single on-chain transaction, the blue boxes are on-chain, the orange boxes visualize the intermediaries executing the transactions and the dashed lines symbolize the begin and the end of the callback transaction. *Intermediary Init* starts the transaction by importing the transaction values to the invocation contract (step 1), which invokes *Method A* (step 2). *Method A* invokes an external method via the distribution contract (step 3) and returns the received invocation id from the distribution contract with the status pending to the invocation chain (steps 4 and 5). After the distribution contract has received a result (step 6 and 12), the callback transaction is executed (step 7 and 13). Depending on whether the method invokes a different blockchain again (*Method B*, steps 8-10) or returns a result (*Method C*, steps 13 and 14), the status is updated differently. Either the pending transaction on the invocation contract is updated with the new invocation id (*Method B*, step 11) similar to *Method A* or the final result is received (*Method C*, step 15). The final result is returned to *Intermediary Init* (step 16).

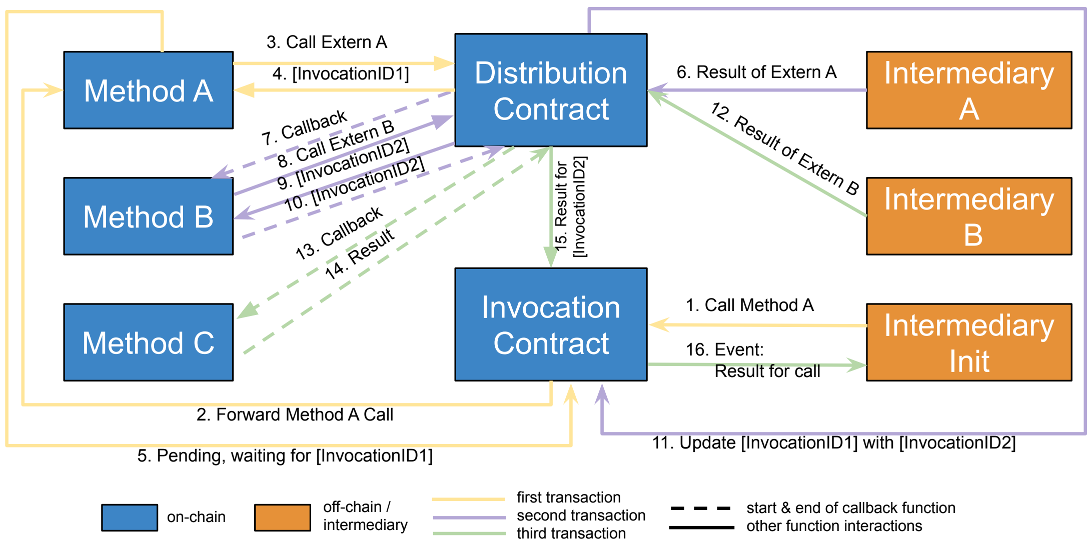


Figure 4.4: Chain of Smart Contract Calls

Recursive Execution Attack. Theoretically, an attacker can create an endless chain (imagine an infinite loop as shown in Figure 4.5 where the contracts of two chains call each other) so that the transaction status never reaches its final state as long as enough gas is provided by the attacker. This harms the intermediary, since it does not get its tokens for the transaction back while the loop continues. If the intermediary discontinues its services during the "endless" waiting period, the transaction is considered as fraud and the paid gas will not be returned. The attack is particularly interesting for token transfers, which are described in the following sections, where an intermediary has to prepay the tokens for a caller.

To limit the transactions between recursive systems, solutions like a maximum call depth can be used. For this, the intermediary has to transfer the current call depth to the other blockchain. Furthermore, the amount of callback function has to be limited to prevent infinite continuations on a single chain. Beside limiting the execution count and transmitting the depth value to the target chain, no satisfactory solution for an endless chain has been found so far. Thus, no anti-recursion patterns are included in the design and this thesis assumes that in the worst case, the transaction ends when there are no more tokens available.

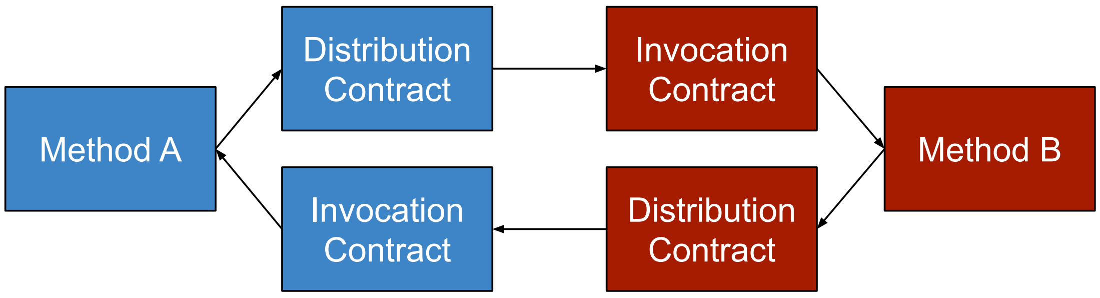


Figure 4.5: Recursive Execution Attack

4.3.10 Extension: Sending Tokens to a Smart Contract

Not only parameters have to be sent to the smart contract, but it may also happen that tokens have to be transferred. Therefore, an additional field is added in the `registerCall` function that indicates the amount of tokens to be paid to the contract on the target chain. The payment has to be additionally checked by the validators.

Since the value of one token is not equal between two chains, an exchange rate has to be used. In addition, we define that the number of tokens is registered in the currency of the target chain, since the receiving amount is supposed to be static (e.g., a merchant expects to get the tokens exactly).

There are several solutions for finding a suitable exchange rate:

- **Exchange rate is managed by contracts**
  The current exchange rate is either stored in the contract of the source chain or the target chain. The update process could be solved similar to the gas price updates of the target blockchain (see Section 4.3.5). Unlike the gas price, it does not matter where the exchange rate is stored, since each blockchain pair has a unique conversion value. Since billing takes place in the source chain, the distribution contracts of both chains contain the exchange value. However, this approach results in many adjustment transactions because the market value fluctuates.

- **Exchange rate will be proposed during the offer phase**
  The intermediaries propose an exchange rate while making an offer to the caller. In addition, the caller defines a maximum exchange rate that prevents excessive offers. Here, at least $max\ exchangerate \cdot tokens$ tokens have to be deposited for the token transfer. This option is preferred, since no additional exchange rate updates

need to be made. The best offer is then calculated by adding the exchange rate to the formula:

$$\text{price} = \text{gasprice} \cdot \text{startgas} + \text{max exchangerate} * \text{tokens} \tag{4.4}$$

The deposit of the caller while registering the call is calculated as follow:

$$\begin{aligned} &\text{max gasprice} \cdot \text{startgas} + \text{fees} \\ &+ \text{callback startgas} \cdot \text{callback gasprice} \\ &+ \mathbf{max\ exchangerate} \cdot \mathbf{tokens} \leq \text{deposited tokens} \end{aligned} \tag{4.5}$$

4.3.11 Extension: Receiving Tokens from a Smart Contract

Unlike sending tokens, this type of transaction is more difficult to solve because of following issues:

- **The caller may want to specify a minimum exchange rate compared to sending tokens.**
  This can be easily solved by defining this parameter as well. This parameter limits the exchange rate between a minimum and a maximum value when tokens are sent and received. In addition, the formula for the best offer has to be adjusted again:

  $$\text{price} = \text{gasprice} \cdot \text{startgas} + (\text{sent tokens} - \text{received tokens}) \cdot \text{exchangerate} \tag{4.6}$$

- **The intermediary receives the tokens first and has to forward them to the source chain contract.**
  If the intermediary behaves dishonestly, the tokens are lost. Therefore, the caller has to specify the maximum number of tokens to be withdrawn and the intermediary has to deposit at least the amount of tokens at the specified exchange rate.

- **The source chain contract has to ensure that the tokens can be withdrawn.**
  This can be achieved by forwarding the tokens to the callback function or the initiator of the cross-chain transaction.

- **The execution can be executed by someone else.**
  In this case, the tokens are lost. Since the transaction is executed via a contract, the execution can be tied to this contract. However, this does not solve the problem that the contract has to forward the tokens to the dishonest intermediary. Therefore, the payout transaction has to be tied to the winning intermediary. This requires an additional method call by the caller. Admittedly, the caller has to observe the blockchain to perform this action, which has heretofore been omitted to reduce the complexity for the caller.

  An option that does not require the caller, is to split the import step. First, the intermediaries import the transaction into the target chain where it is checked by the validators. Only if the import is successful, the intermediary can execute this transaction in a second step. This introduces a voting on the target chain and requires an additional call from the intermediary to execute the transaction (one for registering and one for executing the transaction). Since this option does not require any observation by the caller, it is preferred. However, the time to complete a cross-chain transaction increases as additional voting is required. The voting fee is paid in advance by the intermediary (as the caller may not have an account on the target chain), which will be refunded by the caller at the end of the transaction if the intermediary is acting honestly.

Hint 4.6 *Due to limitations with the representation of decimal numbers in the Solidity language, token transfer was not implemented in the prototype. For more details, see the corresponding chapter in the implementation section (Section 5.6).*

4.4 Conceptual Architecture

This section combines the discussed methods as an architecture for invoking smart contracts across blockchains. Figure 4.6 shows the interactions between two blockchains without token transfers and the deposition and withdrawal of tokens. Table 4.1 summarizes the introduced functions. The protocol follows the steps given in Figure 4.6 and is summarized as follows:

1. The caller registers the cross-chain smart contract invocation.

2. Intermediaries make offers.
   Optional: If the caller wants to cancel the transaction, it can abort it.
   Optional: Intermediaries can revoke their offers.

3. An offer of an intermediary is accepted. The selected intermediary imports the transaction data into the invocation contract and executes the transaction. If the intermediary is not delivering the results, the voters can vote for a dishonest intermediary.

   The yellow arrows illustrate a possible recursive cross-chain call:

   a) The callee registers a cross-chain invocation at the distribution contract.

   b) Once the result is received, the distribution contract calls the invocation contract with the return value for updating the status of the transaction.

4. The intermediary listens to the transaction status.

5. Once the status has been set to completed, the intermediary will import the value into the distribution contract on the source chain.

6. The validators verify the forwarded data and vote whether the intermediary behaves correctly.
7. The transaction is finalized by the intermediary after a positive vote. If the vote was negative, the caller or the winning validator invokes the `finalize` function to retrieve the tokens.
8. The caller fetches the result of the cross-blockchain call.

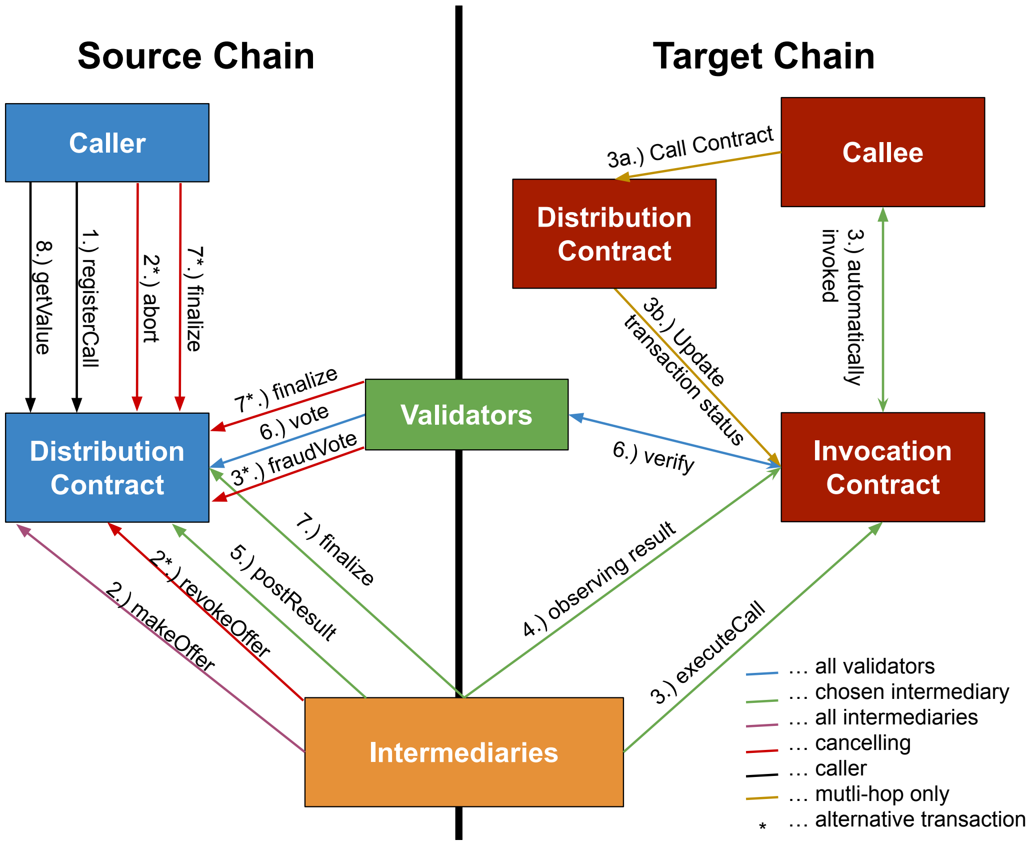


Figure 4.6: Interactions between Caller, Distribution Contract, Intermediary, Invocation Contract, Callee and Validator

Table 4.1: Methods Used for Calling Smart Contracts across Blockchains

Name	Initiator	Description
`registerCall`	Caller	A caller registers a cross-blockchain function call.
`abort`	Caller	A caller can abort its registered transaction as long as no offer has been accepted.
`depositTokens`	All	A participant deposits tokens as guarantee.
`withdrawTokens`	All	A participant revokes its deposit.
`makeOffer`	Intermediary	An intermediary is making an offer for executing the registered transaction.
`revokeOffer`	Intermediary	As long as an offer has not been accepted, the offer can be revoked.
`executeCall`	Intermediary	The intermediary is executing the registered call on the target blockchain.
`postResult`	Intermediary	The intermediary is submitting the result received from `executeCall`.
`fraudVote`	Validator	The validator votes for a fraudulent behavior of the intermediary.
`vote`	Validator	The validator votes on the submitted result whether it is correct.
`finalize`	All	Someone (mostly Intermediary) finalizes the external execution, continues with the next function and distributes the tokens.
`getValue`	Caller	The caller receives the returned value.

4.5 Discussion

This section introduced a design for calling smart contracts between blockchains. In each section, several approaches were discussed. The proposed solution tries to a find a good compromise between security, cost and usability. It pays attention to block consistency by waiting a certain amount of time before proceeding to the next step, and uses a voting algorithm for selecting the intermediary as well as for verifying the successful execution on the target blockchain. Compared to cross-blockchain tokens, it is more important to focus on gas cost by avoiding redundant work because the gas cost for executing an unfamiliar smart contract function can be much higher. In addition, the voting mechanisms are in most cases kept simple and are not endowed with security features like the Merkle proof used in Metronome, although such concepts could be added in a later step to eventually improve the stability of the voting algorithms. Furthermore, concepts, like the usage of incentives for defining the gas rate or the exchange rate have been introduced, and more complex topics such as chain continuation or cross-blockchain user restriction have been discussed. The proposed solution meets all the requirements introduced in the motivational scenario and all the use cases mentioned in Section 4.2. In the following subsections, these requirements are discussed in detail. The given architecture will be implemented in Chapter 5.

4.5.1 Requirement Evaluation

1. **Invoking smart contracts on a different blockchain.**
   This is possible with the proposed architecture by following the protocol introduced in Sections 4.3 and 4.4.

2. **Scalability.**
   The solution is scalable by increasing the number of validators and intermediaries. In addition, different intermediaries and validators can be used for each blockchain combination. Furthermore, it is possible to duplicate the smart contracts to distribute the load between the systems.

3. **Security.**
   Since the majority of votes are required to finalize a transaction and the rules include a verification of all parameters and return values, the security of this architecture is guaranteed as long as at least 50% of the validators behave honestly. In addition, a method for using nonces and asymmetric cryptography for one-time method invocations on behalf of the caller was discussed, which can be used if a repeated execution is forbidden.

4. **Consistency.**
   The block wait time for further steps has been taken into consideration while planning the protocol. Therefore, the proposed protocol can handle eventually consistent data.

   The handling of blockchain forks was not discussed, as intermediaries and validators have to decide which blockchain to follow. This is most likely done in an own blockchain as discussed in Chapter 4.3.3. In most cases, some transactions will fail in the fork detection period, since some voters or intermediaries will use different versions of the blockchains for executing or validating the transactions.

4.5.2 Use Case Evaluation

1. **Restrict the function access to specific users.**
   The protocol itself does not support restricting access to specific callers. However, a method has been discussed which can be implemented by the developer.

2. **Sending and receiving tokens.**
   A concept for the transfer of tokens between different blockchains has been introduced. It uses an exchange rate provided by the intermediary during the offer. The concepts were described in Sections 4.3.10 and 4.3.11.

3. **Chain of smart contract calls.**
   The architecture supports to call a smart contract across blockchains within a smart contract of the target chain as discussed in Section 4.3.9. Therefore, this use case is supported by the architecture.

4. **Continue with the execution after invoking a smart contract.**
   The architecture supports the continuation of a transaction when calling the `finalize` function. Thereby, a previously saved function is called with the return value as discussed in Section 4.3.8.

CHAPTER 5

Implementation

This chapter describes the implementation of the architecture presented in Chapter 4. First, in Section 5.1, a brief overview of the technology stack is given. Then each component is presented in a separate section: Invocation Contract (Section 5.2), Distribution Contract (Section 5.3), Intermediary (Section 5.4), and Validator (Section 5.5). The chapter concludes with a brief discussion about limitations of the prototype (Section 5.6).

5.1 Technology Stack

As discussed in Section 4.1, the Ethereum blockchain is used for the prototype implementation. Thus, Solidity is used as programming language for the smart contracts. There are several ways to write and compile Solidity code. One possible option is to use the official Ethereum Remix IDE[1], which is provided as a Web-based solution and includes a security analysis option. The security analysis option allows developers to check the code for common mistakes like potential reentrancy bugs, the usage of tx.origin, block timestamp and blockhash in functions as well as the correct usage of selfdestruct. Another option is to use the command-line compiler *solc*, which can be controlled using, amongst others, the JavaScript library *solcjs*[2].

For local development, there are also frameworks that facilitate the work. One is the *Truffle Suite*[3], which provides *Ganache*, a local Ethereum blockchain for development, and *Truffle*, a local development environment. However, in this work, *Truffle* was not used to develop the prototype because we decided against an additional tool.

Furthermore, we use *Docker*[4] to easily set up and run various Ethereum test networks using *Ganache*. This allows a simple simulation of the architecture.

[1] `https://remix.ethereum.org`, accessed 2019-04-26
[2] `https://www.npmjs.com/package/solc`, accessed 2019-04-26
[3] `https://truffleframework.com/`, accessed 2019-04-26
[4] `https://www.docker.com/`, accessed 2019-04-26

For intermediaries and validators who need to process and transfer information between multiple blockchains, the JavaScript Ethereum API *web3.js*[5] is used. The code is executed using *node.js*[6], a JavaScript runtime environment.

Alternatively, there is also *Web3.py*[7], an alternative library for Python-based applications for interacting with Ethereum, based on *web3.js*.

In summary, the following technology stack is used:

- Solidity + solc + solcjs
- Docker + Ganache
- web3.js + node.js

5.2 Invocation Contract

The invocation contract is responsible for tracking and logging the call on the target chain. For this purpose, the invocation contract includes a method named `executeCall` that stores and logs all execution parameters and results. Furthermore, the contract includes methods that update the progress of the result if a call across multiple blockchains is required (see Section 4.3.9).

5.2.1 Execute Call

Each time `executeCall` is called, a new result entry is created that contains information about the transaction (gas price, amount of steps), the call (result status, result data, parameters, contract address, maximum gas, unique call id) as well as information needed by the intermediary and the validators (source chain, sender address, current block number). This entry is used by the intermediary to transfer the data back to the source chain and by the validators to verify that the intermediary has used the correct data.

5.2.2 Update Invocation Status

In some cases, the result is not immediately available (e.g., a transaction is calling another blockchain again). Therefore, the contract includes the ability to publish the final return result at a later time by setting the return status to pending. In order for this return status to be recognized by the invocation contract, the callee must notify the contract. This is done by using a hard-coded 32-byte keyword as return value (0xcafefeedcafefeedcafefeedcafefeedcafefeedcafefeedcafefeedcafefedcafefeed).

Subsequently, the invocation contract registers a listener at the distribution contract to get notified when an update is received. There are two different methods to receive the

[5]`https://web3js.readthedocs.io/en/1.0/`, accessed 2019-04-26
[6]`https://nodejs.org/en/`, accessed 2019-04-26
[7]`https://web3py.readthedocs.io/en/stable/`, accessed 2019-04-26

update: `updateResult` and `finalResult`. While `finalResult` only updates the result entry with the data, `updateResult` analyzes the result data and checks if the keyword regarding a cross-blockchain call is present again to perform a new registration for an update event. For security reasons, both functions can only be invoked by the distribution contract while checking for possible registered listeners. The decision which function is used, is made by the distribution contract and is described in Section 5.3.

When one transaction registers multiple cross-blockchain calls in one function, it is assumed that the last registered cross-chain transaction is always used to update the return value and return status. The assumption is based on the fact that normally the return statement is always the last statement of a function.

This has to be taken into account when implementing callee contracts. Listing 5.1 shows parts of the described methods (verifying the keyword, registering a listener and returning the keyword) for better understanding:

- Verifying the keyword:
  - Line 2: Definition of the keyword.
  - Line 5: Gets the last `invocationId` before invoking the callee contract.
  - Line 6: Calls the callee contract.
  - Line 7-11: If the invocation was successful (line 7), the `invocationId` is increased, i.e., a cross-blockchain call has been triggered in the callee contract (line 8) and the keyword matches the defined keyword from line 2 (line 9), then a listener for the result of the cross-blockchain call is registered (line 11).
  - Line 12: Sets the execution status to `ExecutionWaiting` so that the intermediary knows that the result is not ready and should not be published on the source chain.
- Registering a listener:
  - Line 19: Verifies whether the invocation contract calls this function.
  - Line 20: Stores the result id received from line 11 to notify the invocation contract as soon as the last generated `invocationId` (=numCallings) has received its result.
- Returning the keyword:
  - Line 25: Invokes a cross-blockchain call.
  - Line 26: Returns the keyword so that the invocation contract waits for the result of the cross-blockchain call.

Listing 5.1: Example of a Register Call

```
// Invocation Contract
bytes waitingCode = hex"cafefeedcafefeedcafefeedcafefeedcafefeedcafefeedcafefeedcafefeed";

function executeCall() {
  uint tmp = distributionContract.getNumCallings();
  (bool success, bytes memory resultData) = contractAddress.call(...);
  if (success &&
    distributionContract.getNumCallings() > tmp &&
    waitingCode.equalStorage(resultData)
  ) {
    distributionContract.registerNotifyOnResult(resultId);
    status = ResultStatus.ExecutionWaiting;
  }
  ...
}

// Distribution Contract
function registerNotifyOnResult(uint resultId) external {
  require(msg.sender == address(invocationContract), "Only invocationContract can call this
      function");
  listeners[numCallings] = resultId;
}

// Example Contract
function foo() public returns (bytes32) {
  distributionContract.registerCall(...);
  return hex"cafefeedcafefeedcafefeedcafefeedcafefeedcafefeedcafefeed";
}
```

Hint 5.1 *According to the concept presented, the use of the return value "0xcafefeed-cafefeedcafefeedcafefeedcafefeedcafefeedcafefedcafefedcafefed" is allowed if there is no triggered cross-blockchain call. Otherwise it is assumed that the result of the external call should be used as the return value.*

In the highly unlikely event that the return value should match the keyword and an external call is triggered, a continuation function (see Section 5.3) can be used which will be executed after the external call has been returned. This function offers the possibility to update the return value with the keyword.

Since transactions are executed sequentially, a more elegant way to solve this issue is to let the smart contract inform the invocation contract by a method call of an additional external call. However, the address of the invocation contract has to be known in advance by the contract, if `msg.sender` *cannot be used (subcontract). Thus no exchange of the invocation contract is possible without updating all depending contracts. Therefore, the decision was made to use the keyword.*

5.3 Distribution Contract

The distribution contract manages the phases of a cross-blockchain call, offering and voting processes and the continuation of smart contracts.

5.3.1 Phases of the Distribution Contract

The transitions of the phases of the distribution contract are based on block numbers and published data, since a block can be considered as part of the chain with high probability after a certain number of subsequent blocks. The following list describes the conditions under which the cross-blockchain call is in a particular phase.

1. **PreOfferPhase**: This phase is valid once a call has been registered and ends when `waitingBlocks` have passed to ensure that the call registration has been successfully added to the chain.

2. **OfferPhase**: This phase follows the *PreOfferPhase* for `blocksPerPhase` blocks. At this stage, intermediaries have the opportunity to submit offers.

3. **PreTransactionPhase**: Similar to the *PreOfferPhase*, this phase is a buffering phase of `waitingBlocks` blocks to ensure the correct winner is selected. It starts immediately after the *OfferPhase*.

4. **TransactionPhase**: This phase starts immediately after the end of the *PreTransactionPhase* and lasts until the final result is published.

5. **PreVotingPhase**: With the release of the final result, this phase begins for `waitingBlocks` blocks to ensure that the result has not been changed.

6. **VotingPhase**: This phase begins after the *PreVotingPhase*. During this period, the validators have the opportunity to vote whether the published result is correct. The validators have `blocksPerPhase` blocks time to submit their voting after the first vote has been received. This ensures that at least one voting has always been registered before the end of the phase.

7. **PostVotingPhase**: This phase starts immediately after the *VotingPhase* and is again a buffer of `waitingBlocks` blocks to ensure that all votes have been successfully recorded and processed.

8. **WaitForFinalization**: This phase begins after the *PostVotingPhase* and ends when the `finalize` function is called to complete the call.

9. **Finalized**: This phase starts after *WaitForFinalization* and remains active until the end of life of the distribution contract.

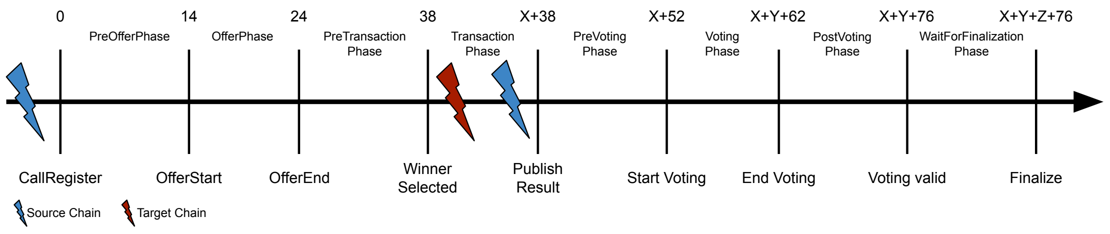


Figure 5.1: Execution Phases

The diagram in Figure 5.1 shows the phases on a timeline and the number of blocks required to reach each phase when `waitingBlocks` is set to 14 and `blocksPerPhase` is set to 10 blocks. X is the number of blocks while waiting for the result, Y is the number of blocks between the begin of the voting phase and the first vote, and Z is the number of blocks that have passed in the *WaitForFinalization* phase.

Example 5.1 *Assume that the next step is published immediately after the phase has been reached and the transaction result is considered as valid after 14 blocks. Then,* $Y = Z = 0$ *and* $X = 0 + 14 = 14$ *and the total number of blocks to reach the final phase is* $76 + 14 = 90$*, which corresponds to 23 minutes at a hash rate of four blocks per minute.*

5.3.2 Register Call

Once a caller wants to execute a cross-blockchain transaction, it has to register the call. Thereby, the following parameters have to be passed:

- **blockchainId**: An id (4 bytes) that uniquely identifies a target blockchain.
- **contractId**: The address of the callee
- **parameters**: Contains all data which are passed on directly to the callee. The data value contains the method signature and the parameters encoded in a bytes datatype.
- **maxSteps**: The maximum number of steps that can be used to execute the smart contract.
- **maxGasPrice**: The maximum gas price that can be submitted for an offer.
- **callbackSteps**: The maximum number of steps that can be used when a smart contract is continued. 0, if no continuation is required.
- **callbackAddress**: The contract address, which should be called upon continuation. 0, if no continuation is required.

Listing 5.2: Example of a Register Call

```
1 DistributionContract distributionContract;
2 address targetContractAddress;
3
4 function foo() public {
5   bytes memory params = abi.encodeWithSignature("setText(string)", "Hallo Welt");
6   bytes targetContract = abi.encodePacked(targetContractAddress);
7   distributionContract.registerCall.value(3000000000000000)("ETH0", targetContract, params,
        50000, 9999, 0, address(0), "");
8 }
```

- **callbackMethod**: The name of the method to call when continuing. Empty if no continuation is required.

The `registerCall` method creates a unique `invocationId` that is returned and published with an event to the intermediaries and validators. This id identifies the cross-blockchain call and is needed to track the status. Listing 5.2 shows an example of correctly invoking the `registerCall` function with its arguments. In line 5, the function name with its arguments is encoded and in line 6 the callee contract address is converted to bytes. The blockchain *ETH0* in invoked in line 7 with the encoded address, the encoded function name as well as parameters with a maximum of 50000 steps, a maximum gasprice of 9999 per step and no callback function. For this, 3,000,000,000,000,000 Wei (=0.003 Ether) are deposited to the distribution contract for the payment of the cross-blockchain transaction.

5.3.3 Offering Process

The offering phase selects the winner based on the signature and the best offer (gasPrice). Each offerer has to ensure for itself that the gas price is chosen so that it does not incur a loss due to a bad exchange rate. Furthermore, the offerer has to ensure that the selected gas price on the target chain is high enough for the transaction to be executed on time. If this is not the case, the offerer is classified as a fraud. In order to reduce the number of frauds, each offerer has to deposit tokens in advance. These tokens are consumed when the winning offerer cannot publish the results in time to pay the fees for the validators and to compensate the caller.

5.3.4 Voting Process

In general, the voting process is divided into multiple phases, whereby the voting phases correspond to the phases of the distribution contract (*PreVoting*, *Voting*, *PostVoting*, *WaitForFinalization*, *Finalized*). With each voting, the voter with the lowest signature and the correct voting result wins the voting and receives a voting fee. The voting process is used for result voting and fraud voting. While the result voting verifies the result published by the intermediary, the fraud voting complements the transaction phase in which an intermediary may cause harm to the caller by failing to complete its task.

The fraud voting is performed according to the following rules: The result of a transaction of a cross-blockchain call must be registered within `fraudDistanceBlocks` blocks so that the execution is guaranteed and can be verified by the validators. If the result is not available (e.g., multi-chain calling), the result id must be published. The validator has to vote for fraud if the result id was not published or the final result was not returned after it became available within `fraudDistanceBlocks` blocks.

Hint 5.2 *Note that a transaction is considered fraudulent, even if the result id may not be available on time. Thus, in a more complex protocol, validators could observe the transaction throughput on the target chain and dynamically adapt the duration while the result id can be registered by the intermediary.*

As discussed above, there is a difference between the result and the fraud voting. In order to participate in fraud voting, a deposit of tokens has to be made. If a fraud voting is started with the information that it is fraudulent and this information is incorrect, then the voting fee will be paid by the winning candidate of the wrong voting.

5.3.5 Continuation of Smart Contracts

When registering the cross-blockchain call, the caller can pass a callback function that is invoked during the `finalize` function. The function signature must accept exactly one parameter, the invocation id. The developer can use this id to get more information about the result by interacting with the distribution contract. An example is given in Listing 5.3 where the received result (line 4) is published as an event (line 6), if the result is valid (line 5).

If an additional cross-blockchain call has been triggered (as described in Section 5.2), it informs the invocation contract of a result update. Depending on whether a callback function has been defined, the return value of the callback function or the value obtained from the target chain is forwarded to the invocation contract. Since the continuation method can only trigger a cross-blockchain call, but not the result of the target chain, different methods are called to pass the result. In the case of a cross-blockchain call, `updateResult` is called, in the other case `finalResult`.

Hint 5.3 *If the original value of the target chain needs to be returned, but a callback function is defined, then the continuation method must return the original result so that it can successfully be passed to the invocation contract.*

Listing 5.3: Example of a Callback Function

```
event CallbackEvent(uint id, bytes);

function callback(uint invocationId) public {
  (bool valid, bool resultStatus, bytes memory result) =
       distributionContract.getValue(invocationId);
  if (valid && resultStatus) {
    emit CallbackEvent(invocationId, result);
  }
}
```

5.4 Intermediary

The intermediary has the task of listening to new calls, registering offers, executing all won offer contests, publishing the return values to the source chain, and finalizing the transactions. For this purpose, the intermediary subscribes to all participating blockchains and listens for events. After an event is received, certain operations are performed. The following subsections summarize these.

5.4.1 Offering

As soon as the intermediary receives a call event, it periodically checks the current phase of the call. Once the *OfferPhase* is reached, the intermediary publishes an offer where the gas price per step is lower than the current offer. When a new offer takes the lead, the intermediary receives this information through a status update. If the first place is lost, a new offer will be made, if the exchange rate permits. Otherwise, the intermediary waits until the *TransactionPhase* is reached.

Hint 5.4 *The current prototype implementation does not include a complex protocol that takes into account timing decisions or signature values when submitting an offer. The intermediary always submits a value below the current offer, as long as a certain threshold has not been exceeded.*

5.4.2 Executing Transaction

The intermediary calls the invocation contract with the specified parameters received from the distribution contract on the target blockchain. After that, the intermediary waits for an event of the invocation contract, in which an unique id is published, over which further information, such as the result, can be retrieved. The intermediary submits this id to the source chain to register the execution of the cross-blockchain call. Once the final result is available, the result is transferred to the source chain.

5.4.3 Finalizing Transaction

Once the voting is over, the intermediary calls the `finalize` function to receive the tokens spent in advance to execute the transaction. For this, the intermediary has to ensure that enough gas is provided for the callback function specified by the caller.

Hint 5.5 *In the prototype, the intermediary or the contract do not check if the gas limit has been set by the caller within an acceptable range. Therefore, this could be an attack vector that prevents the intermediary from successfully receiving its fees due to an "out of gas" error.*

In addition, the prototype assumes that the transaction is valid and does not verify the voting result before calling the `finalize` function.

5.5 Validator

The validator is responsible for verifying the results and detecting fraud.

5.5.1 Result Verification

Once a result is available and the voting begins, the validator compares the result stored on the target chain with the result stored on the source chain. If these values coincide, "valid" is submitted, otherwise, "invalid". As discussed in Section 4.3.6, there is an additional status if the result is valid but the required gas does not match. Listing 5.4 shows an excerpt of the verification script that visualizes the parameters that are verified at each level (0 = invalid, 1 = valid, but gas invalid, 2 = valid):

- Line 2-3: Defines the parameters for each level that have to be checked.
- Line 4-8: Verifies that the level 1 parameters match between the source and the target chain. If not, invalid will be returned.
- Line 9-13: Verifies that the level 2 parameters match between the source and the target chain. If not, valid, but gas invalid will be returned.
- Line 14: Returns valid if all parameters match.

The following list describes the reason for the verification of each parameter.

- **invocationId**: If the same request is called twice by the caller, the intermediary has to execute it twice and cannot reuse an old result.
- **parameters**: The intermediary has to pass the correct parameters to the target chain, otherwise they will not match the requested execution behavior.
- **resultStatus**: The result status defines whether the transaction has been executed successfully. Returning the wrong value would lead to an incorrect assumption about the transaction status on the target chain.
- **result**: The result contains the return value of the transaction and would be used incorrectly in later functions, if this value does not match.

Listing 5.4: Execution Validation

```
function validateCall(source, target) {
  let level1 = ['invocationId', 'parameters', 'resultStatus', 'result', 'contractId',
      'maxSteps'];
  let level2 = ['requiredSteps'];
  for (el of level1) {
    if (source[el] != target[el]) {
      return 0;
    }
  }
  for (el of level2) {
    if (source[el] != target[el]) {
      return 1;
    }
  }
  return 2;
}
```

- **contractId**: This field contains the address of the callee contract to verify that the correct contract has been invoked.
- **blockchainId**: This field is checked implicitly by retrieving the results from the blockchain submitted by the caller. It ensures that the intermediary has submitted the execution to the correct chain.
- **maxSteps**: If the maximum amount of steps specified by the caller do not match, the contract may fail due to an unpredictable "out of gas" error.
- **requiredSteps**: If the intermediary submits more steps than is necessary to execute the transaction, it tries to obtain more tokens than it is entitled to.

5.5.2 Fraud Detection

The validator observes the blockchain for events and checks whether the intermediary is executing the transaction correctly. Thereby, the validator checks whether the intermediary has submitted the result id and the result value, if available, on time. If a fraud is detected, the validator will participate in a fraud voting. The cross-blockchain call is finalized by the winner of the voting to receive its reward.

5.6 Limitations of the Prototype

Since the prototype is a proof of concept, there are some limitations, which are explained in detail in the following sections.

5.6.1 Stack Depth and Local Variables

The number of local variables within a function is limited by the stack size. Therefore, the use of modifiers sometimes results in errors because the maximum number of variables is reached earlier. Therefore, the code of the modifiers has been written inside the methods and modifiers have not been used at all. As a side effect, this has the advantage that the security analysis offered by the Remix IDE works better, since modifiers cannot be analyzed for re-entrance attacks.

5.6.2 Token Transfer

Since the Solidity language does not support decimal numbers at the moment, the concept of the exchange rate as discussed in Section 4.3.10 cannot be used.

An alternative way would be to support fractions by passing numerators and denominators individually. However, this leads to increased complexity and even more variables on an already limited stack size. Therefore, the token transfer has been dropped and can be added later if needed.

5.6.3 Stale Blocks

The intermediary and the validator do not check the events for stale blocks for simplicity and readability of the source code. Otherwise, all events have to be stored and verified to see if the status has not changed after n blocks.

5.6.4 Other Limitations and Peculiarities

In addition to the limitations stated above, the current implementation has the following limitations.

- The return value is always taken from the last cross-blockchain call, instead of letting the callee decide which return value should be used.
- The keyword cannot be returned when a cross-blockchain call is triggered (see Section 5.2.2).
- The result of the callback function is always used to update the result data instead of letting the caller decide whether the return value is deepening on the callback function.
- The intermediary always selects a lower gas price instead of taking the signature and the market value into consideration.
- If no intermediary has participated in the offering process, the prototype does not mark the cross-blockchain call as finished according to Section 4.3.6. Therefore, the caller has to observe the blockchain to finalize the cross-blockchain call and receive back its deposit, if no offer has been submitted.
- The intermediary finalizes all offers it has won, independently of the voting result.
- The gas value of the callback function is not verified as described in Section 5.4.3.

- There are no restrictions regarding the recursive execution attack.
- The prototype (intermediary, validator) has no error handling and retry mechanism if an transaction has not been processed successfully.
- The prototype does not handle an id overflow error.
- From the point of game theory, the validator may not vote honestly but by majority to receive a compensation as discussed in Section 4.3.5.

CHAPTER 6

Evaluation

This chapter evaluates the prototype presented in Chapter 5. First, the evaluation setup is specified in Section 6.1. Then, several evaluation scenarios are given in Section 6.2.

6.1 Evaluation Setup

The evaluation setup specifies the infrastructure used for the following scenarios. As defined in Section 5.1, we again use different Ganache blockchains in Docker containers for the setup. For the considered tests, we define three blockchains, each with a cost of 1 gas per step, so we can easily estimate the actual costs of the live Ethereum blockchain, and with slightly different block times to achieve a different block mining progress between the chains (4-6 seconds). The ports of the containers are forwarded to the localhost. The configurations of the containers are described in Table 6.1 where `blockTime` describes the time after a new block is added and `gasPrice` defines the cost per step.

We choose Ganache as a local test environment because it is easy to use and provides adjustable parameters such as gas cost or block mining time, and do not use official test networks unless specified otherwise, since for most scenarios the evaluation is independent of other transactions in the network. This allows us to spawn and destroy multiple networks as needed for the experiment without being tied to the few existing public test networks, all of which require gigabytes of storage. Furthermore, we can guarantee a controlled environment for the test cases where unexpected failures of the test networks do not affect the test result.

In addition, all blockchains are initialized with the developed contracts. The parameters for `waitingBlocks` and `blocksPerPhase` are both set to 10 unless specified otherwise. The reason behind a relatively low `waitingBlocks` time is that it shortens the test time. However, this is only possible because we know that the blockchains of the test network do not fork and thus the `waitingBlocks` time is not needed at all. All

Table 6.1: Definition of Evaluation Blockchains

Name	LocalPort	Blockchain Settings
ETH0	8510	--blockTime=5 --gasPrice=1
ETH1	8520	--blockTime=6 --gasPrice=1
ETH2	8530	--blockTime=4 --gasPrice=1

Table 6.2: Functions of the TestContract

Function	Description
setBool(bool)	This function stores a boolean value in the contract.
setNumber8(uint8)	This functions stores a 8 bit integer in the contract.
setNumber256(uint256)	This functions stores a 256 bit integer in the contract.
setText(string)	This functions stores an utf8-encoded text in the contract.
setBytes(string)	This functions stores various bytes in the contract.
setAddress(address)	This functions stores an address in the contract.
getBool()	This function returns the stored boolean.
getNumber8()	This function returns the stored 8 bit integer.
getNumber256()	This function returns the stored 256 bit integer.
getText()	This function returns the stored text.
getBytes()	This function returns the stored bytes.
getAddress()	This function returns the stored address.
callbackText(uint)	A function for receiving a text callback. It stores the received data in the own contract and returns the result.
callSetText(string)	This function invokes `setText` on ETH1 and registers `callback1` as callback function.
callback1(uint)	This function calls `getText` on ETH1 and registers `callbackText` as callback function.
callGetText()	This function calls `getText` on ETH1 and does not register any callback function.

accounts initially deposit 1 Ether to the *DistributionContracts* for the execution of the cross-blockchain operations, which is enough Ether for all of the scenarios. Furthermore, we create a *TestContract* for each chain that contains different functions for the following test scenarios, which are listed in Table 6.2.

For the following scenarios, when mentioning *ETH0*, we are referring to the *test smart contract* on blockchain *ETH0* unless stated otherwise. This also applies to *ETH1* and *ETH2* for their respective blockchain.

6.2 Evaluation Scenarios

This section evaluates the blockchain in terms of cost and features. In total, twelve different scenarios are evaluated, of which ten focus only on the functional correctness.

6.2.1 Scenario 1: Cross-Blockchain Call – Transaction Order

Description

This scenario verifies the correct transaction order of a cross-blockchain call. There is only one call at a time, only one intermediary making an offer and only one validator verifying the transaction.

The caller registers a cross-blockchain call from Blockchain *ETH0* to *ETH1*, in which the boolean value is set to `true`. The expected result is a change of the value to `true`, and the order of the transactions should match the order given in Section 4.4. In order to ensure a correct test result, it has to be checked whether the value is set to `false` before the evaluation. If this is not the case, the value can be changed by executing the `setBool` function on *ETH1*.

This experiment only has to be evaluated once, since the sequence of operations is deterministically tied to the phases of the distribution contract.

Result

Listing 6.1 shows the relevant parts of the log output. As shown in line 12, the result matches with the assumption and has been previously successfully reset (Line 3). Furthermore, the sequence of the transactions follows the figure in Section 4.4:

- Line 2: Cross-blockchain call is registered.
- Lines 3-5: Offer submitted and chosen as best offer.
- Line 6: Execution on target chain.
- Line 7: Published result on source chain.
- Line 8: Published voting.
- Line 9: Finalization of cross-blockchain call.

Listing 6.1: Log Output Showing the Order of Transactions

```
11:40:38.607 INFO Current bool value on blockchain ETH1: 'false'
11:40:43.173 INFO New event 'NewCallRequest' on contract distributionContract on blockchain ETH0
    blockNumber: 56 data: {"invocationId":{"_hex":"0x01"}}
11:41:33.334 INFO Submitted offer: 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7 invocationId 1
11:41:38.272 INFO New event 'NewBestOffer' on contract distributionContract on blockchain ETH0
    blockNumber: 67 data: {"invocationId":{"_hex":"0x01"}}
11:41:38.375 INFO First place offer: 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7 invocationId 1
11:43:15.548 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH1
    blockNumber: 72 data: {"sender":"0x21574114b889Ea4Dac00d46b445324fdE3b56EA1",
    "resultId":{"_hex":"0x01"}}
11:44:18.590 INFO New event 'ResultAvailable' on contract distributionContract on blockchain ETH0
    blockNumber: 99 data: {"sender":"0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7",
    "invocationId":{"_hex":"0x01"}}
11:45:04.214 INFO Voted 0x7295BB66b1f6e2b49203FDe339482AD8F86c5b50, invocationId: 1, value: 2
11:46:53.910 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
    blockNumber: 130 data: {"invocationId":{"_hex":"0x01"}}
11:46:54.610 INFO Current bool value on blockchain ETH1: 'true'
```

Listing 6.2: Log Extraction Illustrating the correct Result on the Source Chain

```
1   11:54:04.326 INFO Current text value on blockchain ETH1: 'Hello World'
2   11:54:05.254 INFO New event 'NewCallRequest' on contract distributionContract on blockchain ETH0
        blockNumber: 216 data: {"invocationId":{"_hex":"0x04"}}
3   ...
4   12:00:20.983 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
        blockNumber: 291 data: {"invocationId":{"_hex":"0x04"}}
5   12:00:21.154 INFO Result value on blockchain ETH0: '0x00000000000000000000000000000000
        00000000000000000000000000000020 00000000000000000000000000000000
        0000000000000000000000000000000b 48656c6c6f20576f726c640000000000
        00000000000000000000000000000000', translated: Hello World
```

6.2.2 Scenario 2: Correct Propagation of Return Value

Description

This scenario evaluates the correct forwarding of the return value to the source chain. First, the text is set to “Hello World” using `setText` on *ETH1*, which is verified by calling `getText`. Afterwards, a cross-blockchain call is registered from *ETH0* to *ETH1*, which executes `getText` on *ETH1*. The returned result is then checked on the distribution contract by calling the `getValue` function. It is expected that the received value matches “Hello World”.

Again, only one caller, one intermediary and one validator are used. This experiment only needs to be evaluated once, as the result does not depend on the number of invocations.

Result

Listing 6.2 shows the returned result in line 5. Since the result is encoded in a bytes object, it has to be decoded for human readability. Therefore, the decoded value is given as well.

Listing 6.3: Log of a Parallel Execution of Multiple Transactions

```
 1  11:47:15.797 INFO Current bool value on blockchain ETH1: 'false'
 2  11:47:15.993 INFO Current bool value on blockchain ETH2: 'false'
 3  11:47:19.066 INFO New event 'NewCallRequest' on contract distributionContract on blockchain ETH0
        blockNumber: 135 data: {"invocationId":{"_hex":"0x02"}}
 4  11:47:24.135 INFO New event 'NewCallRequest' on contract distributionContract on blockchain ETH0
        blockNumber: 136 data: {"invocationId":{"_hex":"0x03"}}
 5  ...
 6  11:49:52.047 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH1
        blockNumber: 138 data: {"sender":"0x21574114b889Ea4Dac00d46b445324fdE3b56EA1",
        "resultId":{"_hex":"0x02"}}
 7  11:49:56.812 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH2
        blockNumber: 208 data: {"sender":"0x07ABe1858d3cAeac7d7Da86F57BF79dbDC5fcbCB",
        "resultId":{"_hex":"0x01"}}
 8  11:50:39.566 INFO New event 'ResultAvailable' on contract distributionContract on blockchain ETH0
        blockNumber: 175 data: {"sender":"0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7",
        "invocationId":{"_hex":"0x03"}}
 9  11:50:54.654 INFO New event 'ResultAvailable' on contract distributionContract on blockchain ETH0
        blockNumber: 178 data: {"sender":"0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7",
        "invocationId":{"_hex":"0x02"}}
10  ...
11  11:53:14.998 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
        blockNumber: 206 data: {"invocationId":{"_hex":"0x03"}}
12  11:53:30.094 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
        blockNumber: 209 data: {"invocationId":{"_hex":"0x02"}}
13  11:53:31.001 INFO Current bool value on blockchain ETH1: 'true'
14  11:53:31.016 INFO Current bool value on blockchain ETH2: 'true'
```

6.2.3 Scenario 3: Parallel Execution of Cross-Blockchain Calls

Description

In this scenario, different cross-blockchain calls are executed in parallel. Again, there is only one caller, one intermediary and one validator. This experiment is intended to show that the intermediary and the validator can handle multiple calls at the same time period and do not have to wait until one call is finished. Transactions should not be mixed or lost by the parties. This time, the caller executes the same transaction as in Scenario 1, but sends the call not only to *ETH1* but also to *ETH2*. We only evaluate this experiment once to show that nothing is lost.

Result

Listing 6.3 shows that the invocations are not mixed and are handled in parallel. This is shown by the output of the invocationIds in lines 3 and 6, in which the call to *ETH1* is first registered and executed, followed by lines 4 and 7 with the call to *ETH2*, as well as in lines 8 and 11, in which the result of *ETH2* is first published and finalized, followed by lines 9 and 12 with the result handling for *ETH1*. In addition, it is visible that the execution order of an event does not depend on the registration id, since the call to *ETH2* is finalized first (line 8 vs. 9), although it was registered after *ETH1* (line 3 vs 4).

6.2.4 Scenario 4: Offer Process – Multiple Intermediaries

Description

This scenario evaluates a cross-blockchain transaction where multiple intermediaries participate in the offer process. The experiment setup complies to Scenario 1, except that three intermediaries submit offers. It is expected that the three intermediaries always try to make a better offer if they are not in lead and that only the winner will execute the transaction. Therefore, this experiment will be evaluated at least until two different intermediaries have won the offer to show that various intermediaries can win the offer.

Result

Listing 6.4 shows a typical offer process as expected in the description. After the call has been registered (line 1), all participating intermediaries submit an offer (lines 2-4). Once a new block with a better offer has been mined (lines 5-6, 11-13, 18-20, 25-27 and 32-34), the non-leading intermediaries try to submit a better offer (lines 9-10, 16-17, 23-24, 30-31) as long as the offer process has not ended.

Some interesting findings in the offer process are summarized:

- Lines 2, 6: The offer with the best price was not selected.

  Depending on the order of the transactions processed in a block, for each best offer a new *NewBestOffer* line is written. Furthermore, not all transactions are processed in the same block, so two *NewBestOffer* lines are not shown in a row. Unfortunately it has been observed that Ganache rarely groups the transactions into one block, so many blocks contain only one transaction, and thus two *NewBestOffer* events in the same block are not outputted with high likelihood.

- Lines 10, 17: The first offer submitted by an intermediary is better than the next one.

  Since not all transactions are processed in the same block and intermediaries do not save submitted offers, they always submit a new offer when a leader switch has been announced. This leads to increased gas cost and unnecessary transactions by the intermediaries, which can be avoided by storing the submitted offers.

In addition, Listing 6.5 shows two different outcomes of the offer process, where different intermediaries execute the transaction on the target chain. This is illustrated in lines 3 and 9 by the sender value.

Listing 6.4: Log Extraction of a Offer Process with Multiple Intermediaries

```
12:06:31.977 INFO New event 'NewCallRequest' on contract distributionContract on blockchain ETH0
      blockNumber: 365 data: {"invocationId":{"_hex":"0x05"}}
12:07:22.253 INFO Submitted offer: 0xF4cf86A951Fd057571a85c747C24c0A4Eb8B5E08 invocationId 5 gas
      price 9114
12:07:22.272 INFO Submitted offer: 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7 invocationId 5 gas
      price 9285
12:07:23.194 INFO Submitted offer: 0x127E15EE07C07e5A0b3A47aD6A44a23367450536 invocationId 5 gas
      price 8983
12:07:27.093 INFO New event 'NewBestOffer' on contract distributionContract on blockchain ETH0
      blockNumber: 376 data: {"invocationId":{"_hex":"0x05"}}
12:07:27.231 INFO First place offer: 0xF4cf86A951Fd057571a85c747C24c0A4Eb8B5E08 invocationId 5


12:07:27.465 INFO Submitted offer: 0x127E15EE07C07e5A0b3A47aD6A44a23367450536 invocationId 5 gas
      price 7539
12:07:27.513 INFO Submitted offer: 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7 invocationId 5 gas
      price 6772
12:07:37.172 INFO New event 'NewBestOffer' on contract distributionContract on blockchain ETH0
      blockNumber: 378 data: {"invocationId":{"_hex":"0x05"}}
12:07:37.236 INFO Lost first place: 0xF4cf86A951Fd057571a85c747C24c0A4Eb8B5E08 invocationId 5
12:07:37.346 INFO First place offer: 0x127E15EE07C07e5A0b3A47aD6A44a23367450536 invocationId 5


12:07:37.450 INFO Submitted offer: 0xF4cf86A951Fd057571a85c747C24c0A4Eb8B5E08 invocationId 5 gas
      price 7314
12:07:37.468 INFO Submitted offer: 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7 invocationId 5 gas
      price 6784
12:07:42.211 INFO New event 'NewBestOffer' on contract distributionContract on blockchain ETH0
      blockNumber: 379 data: {"invocationId":{"_hex":"0x05"}}
12:07:42.320 INFO Lost first place: 0x127E15EE07C07e5A0b3A47aD6A44a23367450536 invocationId 5
12:07:42.450 INFO First place offer: 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7 invocationId 5


12:07:42.624 INFO Submitted offer: 0xF4cf86A951Fd057571a85c747C24c0A4Eb8B5E08 invocationId 5 gas
      price 6206
12:07:42.652 INFO Submitted offer: 0x127E15EE07C07e5A0b3A47aD6A44a23367450536 invocationId 5 gas
      price 5621
12:08:02.341 INFO New event 'NewBestOffer' on contract distributionContract on blockchain ETH0
      blockNumber: 383 data: {"invocationId":{"_hex":"0x05"}}
12:08:02.523 INFO Lost first place: 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7 invocationId 5
12:08:02.632 INFO First place offer: 0x127E15EE07C07e5A0b3A47aD6A44a23367450536 invocationId 5


12:08:02.741 INFO Submitted offer: 0xF4cf86A951Fd057571a85c747C24c0A4Eb8B5E08 invocationId 5 gas
      price 5589
12:08:02.754 INFO Submitted offer: 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7 invocationId 5 gas
      price 4098
12:08:12.396 INFO New event 'NewBestOffer' on contract distributionContract on blockchain ETH0
      blockNumber: 385 data: {"invocationId":{"_hex":"0x05"}}
12:08:12.558 INFO Lost first place: 0x127E15EE07C07e5A0b3A47aD6A44a23367450536 invocationId 5
12:08:12.671 INFO First place offer: 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7 invocationId 5
```

Listing 6.5: Log Extraction Showing Different Winners in the Offer Process

```
12:08:12.558 INFO Lost first place: 0x127E15EE07C07e5A0b3A47aD6A44a23367450536 invocationId 5
12:08:12.671 INFO First place offer: 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7 invocationId 5
12:09:05.661 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH1
      blockNumber: 330 data: {"sender":"0x21574114b889Ea4Dac00d46b445324fdE3b56EA1",
      "resultId":{"_hex":"0x04"}}

---

12:16:34.136 INFO Lost first place: 0xF4cf86A951Fd057571a85c747C24c0A4Eb8B5E08 invocationId 6
12:16:34.275 INFO First place offer: 0x127E15EE07C07e5A0b3A47aD6A44a23367450536 invocationId 6
12:17:36.288 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH1
      blockNumber: 415 data: {"sender":"0x28A65438EB59408E5398E8Cba29269eAa4fB3d35",
      "resultId":{"_hex":"0x05"}}
```

Listing 6.6: Log Extraction of Two Different Voting Winners

```
 1  13:00:25.398 INFO balance for 0xe87858a6278747C5B6D13d3D34b8944AC9227F31: 98999999999999847970
 2  13:00:30.448 INFO balance for 0x535F9924928De7A6160cBDBCf9c9ade5Ad437809: 98999999999999897559
 3  13:00:35.484 INFO balance for 0x7295BB66b1f6e2b49203FDe339482AD8F86c5b50: 99003359999999235957
 4  13:02:10.682 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
         blockNumber: 1031 data: {"invocationId":{"_hex":"0x07"}}
 5  13:02:11.182 INFO balance for 0x7295BB66b1f6e2b49203FDe339482AD8F86c5b50: 99003919999999235957
 6  13:02:11.189 INFO balance for 0xe87858a6278747C5B6D13d3D34b8944AC9227F31: 98999999999999847970
 7  13:02:11.194 INFO balance for 0x535F9924928De7A6160cBDBCf9c9ade5Ad437809: 98999999999999897559
 8
 9  ---
10
11  13:13:47.615 INFO balance for 0x535F9924928De7A6160cBDBCf9c9ade5Ad437809: 98999999999999787292
12  13:13:52.648 INFO balance for 0x7295BB66b1f6e2b49203FDe339482AD8F86c5b50: 99003919999999175279
13  13:13:57.712 INFO balance for 0xe87858a6278747C5B6D13d3D34b8944AC9227F31: 98999999999999787292
14  13:15:32.847 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
         blockNumber: 1191 data: {"invocationId":{"_hex":"0x08"}}
15  13:15:33.195 INFO balance for 0x7295BB66b1f6e2b49203FDe339482AD8F86c5b50: 99003919999999175279
16  13:15:33.199 INFO balance for 0xe87858a6278747C5B6D13d3D34b8944AC9227F31: 99000559999999787292
17  13:15:33.204 INFO balance for 0x535F9924928De7A6160cBDBCf9c9ade5Ad437809: 98999999999999787292
```

6.2.5 Scenario 5: Voting Process – Multiple Validators

Description

In this scenario, a cross-blockchain transaction is evaluated, where multiple validators submit a vote. The experiment setup is equal to Scenario 1 except that three validators participate in the voting process. It is expected that depending on the signature, different validators will win and get a reward. Therefore, this experiment is evaluated at least until two different validators have won the vote.

Result

Listing 6.6 shows the balance of the validator accounts after the voting and after the cross-blockchain call has been finalized for two different executions. As shown by the token balances in lines 3 and 5 and lines 13 and 16, two different validators received a reward for participating in the vote.

6.2.6 Scenario 6: Triggering a Callback Function

Description

It may happen that a caller wants to continue a task after the result is available. This experiment is based on Scenario 2, but this time the caller adds the `callbackText` function as continuation method while registering the transaction. The function is shown in Listing 6.7.

Since this function saves the result (line 4), the storage is cleared by setting an empty string before the cross-blockchain call is executed. It is expected, that the local variable matches “Hello World”. The experiment only needs to be evaluated once because the sequence is always the same.

Listing 6.7: Callback Function for Scenario 6

```
function callbackText(uint256 invocationId) public returns(string memory) {
  (bool valid, bool resStatus, bytes memory result) =
      distributionContract.getValue(invocationId);
  if (valid && resStatus) {
    text = abi.decode(result, (string));
  }
  return text;
}
```

Listing 6.8: Log Output Showing a Successful Execution of a Callback Function

```
1  14:06:47.161 INFO Current text value on blockchain ETH1: 'Hello World'
2  14:06:50.102 INFO Current text value on blockchain ETH0: ''
3  14:06:55.217 INFO New event 'NewCallRequest' on contract distributionContract on blockchain ETH0
       blockNumber: 1806 data: {"invocationId":{"_hex":"0x09"}}
4  ...
5  14:13:06.054 INFO New event 'ContinuedResult' on contract distributionContract on blockchain ETH0
       blockNumber: 1880 data: {"invocationId":{"_hex":"0x09"},"success":true}
6  14:13:06.054 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
       blockNumber: 1880 data: {"invocationId":{"_hex":"0x09"}}
7  14:13:06.914 INFO Current text value on blockchain ETH0: 'Hello World'
```

Result

When a continuation of a cross-blockchain call is triggered, a `ContinuedResult` event is fired. This event is shown in Listing 6.8 in line 5. The successful update of the text is displayed in line 7.

6.2.7 Scenario 7: Recursive Cross-Blockchain Calls

Description

If the result of a call is depending on another cross-blockchain call, a chain of calls is required. This scenario simulates the use case by executing two cross-blockchain calls. First, *ETH0* calls `callGetText` on *ETH2*, which executes `getText` on "ETH1". It is expected that the result is received via *ETH2* to *ETH0* and should match "Hello World". Again, to ensure functional correctness, the text on *ETH1* is set in advance to "Hello World" and on *ETH0* to an empty string. This experiment only needs to be evaluated once, as the result is always the same. The described experiment is visualized in Figure 6.1.

Result

The log output in Listing 6.9 shows the corresponding chain:

- Lines 3, 6: Call to *ETH2*.
- Lines 5, 9: Call to *ETH1*.

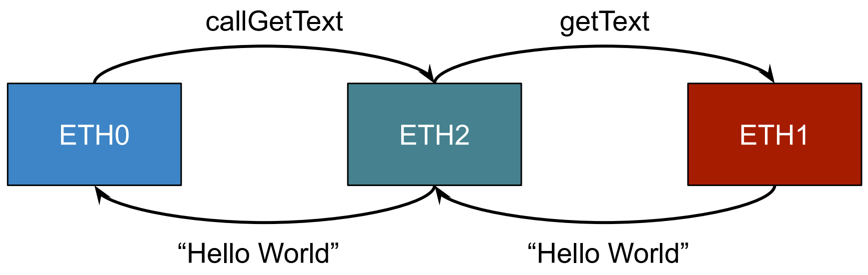


Figure 6.1: Visualization of the Chain of Scenario 7

Listing 6.9: Log Illustrating a Recursive Cross-Blockchain Call

```
1  14:23:48.940 INFO Current text value on blockchain ETH1: 'Hello World'
2  14:23:52.911 INFO Current text value on blockchain ETH0: ''
3  14:23:57.990 INFO New event 'NewCallRequest' on contract distributionContract on blockchain ETH0
        blockNumber: 2010 data: {"invocationId":{"_hex":"0x0a"}}
4  ...
5  14:26:32.715 INFO New event 'NewCallRequest' on contract distributionContract on blockchain ETH2
        blockNumber: 2551 data: {"invocationId":{"_hex":"0x01"}}
6  14:26:32.716 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH2
        blockNumber: 2551 data: { "sender":"0x07ABe1858d3cAeac7d7Da86F57BF79dbDC5fcbCB",
        "resultId":{"_hex":"0x02"}}
7  14:27:18.340 INFO New event 'ResultRegistered' on contract distributionContract on blockchain ETH0
        blockNumber: 2050 data: {"invocationId":{"_hex":"0x0a"}}
8  ...
9  14:28:37.311 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH1
        blockNumber: 1723 data: { "sender":"0x21574114b889Ea4Dac00d46b445324fdE3b56EA1",
        "resultId":{"_hex":"0x09"}}
10 14:29:41.429 INFO New event 'ResultAvailable' on contract distributionContract on blockchain ETH2
        blockNumber: 2598 data: {"sender":"0x07ABe1858d3cAeac7d7Da86F57BF79dbDC5fcbCB",
        "invocationId":{"_hex":"0x01"}}
11 ...
12 14:31:45.822 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH2
        blockNumber: 2629 data: {"invocationId":{"_hex":"0x01"}}
13 14:31:45.823 INFO New event 'NewExecuteUpdate' on contract invocationContract on blockchain ETH2
        blockNumber: 2629 data: {"sender":"0x07ABe1858d3cAeac7d7Da86F57BF79dbDC5fcbCB",
        "resultId":{"_hex":"0x02"}}
14 14:32:28.891 INFO New event 'ResultAvailable' on contract distributionContract on blockchain ETH0
        blockNumber: 2112 data: {"sender":"0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7",
        "invocationId":{"_hex":"0x0a"}}
15 ...
16 14:35:04.328 INFO New event 'ContinuedResult' on contract distributionContract on blockchain ETH0
        blockNumber: 2143 data: {"invocationId":{"_hex":"0x0a"},"success":true}
17 14:35:04.328 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
        blockNumber: 2143 data: {"invocationId":{"_hex":"0x0a"}}
18 14:35:04.435 INFO Current text value on blockchain ETH0: 'Hello World'
```

- Line 7: *ETH0* gets informed that an additional cross-blockchain call has been registered so that the validators are aware of a longer waiting time.
- Line 10: Result of *ETH1* is imported into *ETH2*.
- Line 13: Result update on *ETH2* for invocation of *ETH0*.
- Lines 14, 16: Result from *ETH2* is imported into the source chain *ETH0* and the callback function is triggered.

6.2.8 Scenario 8: Callback Cross-Blockchain Calls

Description

As described in Section 4.3.9, a callback function can also execute an additional cross-blockchain call. This scenario simulates Figure 4.4 given in the mentioned section. For this, *ETH0* calls *ETH2*, which first executes a `setText` function with the text "Hello World" on *ETH1*. While the cross-blockchain transaction between *ETH2* and *ETH1* is finalized, the first callback function is invoked, which calls `getText` on *ETH1* to receive the latest text. Once the second callback is invoked, the value is stored locally on *ETH2* and is returned to *ETH0*. It is expected that the text on *ETH1* as well as on *ETH2* and *ETH0* is set to "Hello World".

Of course, for a meaningful test, the text values on all chains have to be set to a string other than "Hello World", such as an empty string, before starting the experiment. This experiment only needs to be evaluated once to show the correct propagation of the text values.

Result

Listing 6.10 shows the relevant parts of the cross-blockchain invocation. The result is as expected (lines 28-30) and all desired calls are executed, two for the interaction between *ETH2* and *ETH1* and one for *ETH0* and *ETH2*. The following list gives a brief overview of the relevant parts:

- Lines 4, 7, 8: Call from *ETH0* to *ETH2* with a registration for a longer waiting time of the result.
- Lines 6, 10, 11: Call from *ETH2* to *ETH1*.
- Lines 13-15, 18: Finishing call from *ETH2* to *ETH1* and invoking another call to *ETH1*.
- Lines 16: Updating result for *ETH0* on *ETH2*.
- Lines 18-22: Finishing call from *ETH2* to *ETH1*.
- Lines 23: Updating result for *ETH0* on *ETH2*.
- Lines 24-27: Finishing call from *ETH0* to *ETH2* and invoking callback function.

Listing 6.10: Log Extraction Illustrating Multiple Cross-Blockchain Calls

```
14:47:24.324 INFO Current text value on blockchain ETH2: ''
14:47:26.829 INFO Current text value on blockchain ETH1: ''
14:47:31.123 INFO Current text value on blockchain ETH0: ''
14:47:36.277 INFO New event 'NewCallRequest' on contract distributionContract on blockchain ETH0
     blockNumber: 2293 data: {,"invocationId":{"_hex":"0x0b"}}
...
14:50:09.655 INFO New event 'NewCallRequest' on contract distributionContract on blockchain ETH2
     blockNumber: 2904 data: {,"invocationId":{"_hex":"0x02"}}
14:50:09.655 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH2
     blockNumber: 2904 data: {"sender":"0x07ABe1858d3cAeac7d7Da86F57BF79dbDC5fcbCB",
     "resultId":{"_hex":"0x03"}}
14:50:51.683 INFO New event 'ResultRegistered' on contract distributionContract on blockchain ETH0
     blockNumber: 2332 data: {,"invocationId":{"_hex":"0x0b"}}
...
14:52:15.439 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH1
     blockNumber: 1959 data: {"sender":"0x21574114b889Ea4Dac00d46b445324fdE3b56EA1",
     "resultId":{"_hex":"0x0a"}}
14:53:18.180 INFO New event 'ResultAvailable' on contract distributionContract on blockchain ETH2
     blockNumber: 2951 data: {"sender":"0x07ABe1858d3cAeac7d7Da86F57BF79dbDC5fcbCB",
     "invocationId":{"_hex":"0x02"}}
...
14:55:22.731 INFO New event 'NewCallRequest' on contract distributionContract on blockchain ETH2
     blockNumber: 2982 data: {"invocationId":{"_hex":"0x03"}}
14:55:22.732 INFO New event 'ContinuedResult' on contract distributionContract on blockchain ETH2
     blockNumber: 2982 data: {"invocationId":{"_hex":"0x02"},"success":true}
14:55:22.732 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH2
     blockNumber: 2982 data: {"invocationId":{"_hex":"0x02"}}
14:55:22.733 INFO New event 'NewExecuteUpdate' on contract invocationContract on blockchain ETH2
     blockNumber: 2982 data: {"sender":"0x07ABe1858d3cAeac7d7Da86F57BF79dbDC5fcbCB",
     "resultId":{"_hex":"0x03"}}
...
14:57:27.844 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH1
     blockNumber: 2011 data: {"sender":"0x21574114b889Ea4Dac00d46b445324fdE3b56EA1",
     "resultId":{"_hex":"0x0b"}}
14:58:31.220 INFO New event 'ResultAvailable' on contract distributionContract on blockchain ETH2
     blockNumber: 3029 data: {"sender":"0x07ABe1858d3cAeac7d7Da86F57BF79dbDC5fcbCB",
     "invocationId":{"_hex":"0x03"}}
...
15:00:35.620 INFO New event 'ContinuedResult' on contract distributionContract on blockchain ETH2
     blockNumber: 3060 data: {"invocationId":{"_hex":"0x03"},"success":true}
15:00:35.620 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH2
     blockNumber: 3060 data: {"invocationId":{"_hex":"0x03"}}
15:00:35.621 INFO New event 'NewExecuteUpdate' on contract invocationContract on blockchain ETH2
     blockNumber: 3060 data: {"sender":"0x07ABe1858d3cAeac7d7Da86F57BF79dbDC5fcbCB",
     "resultId":{"_hex":"0x03"}}
15:01:17.814 INFO New event 'ResultAvailable' on contract distributionContract on blockchain ETH0
     blockNumber: 2457 data: {"sender":"0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7",
     "invocationId":{"_hex":"0x0b"}}
...
15:03:53.181 INFO New event 'ContinuedResult' on contract distributionContract on blockchain ETH0
     blockNumber: 2488 data: {"invocationId":{"_hex":"0x0b"},"success":true}
15:03:53.181 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
     blockNumber: 2488 data: {"invocationId":{"_hex":"0x0b"}}
15:03:53.384 INFO Current text value on blockchain ETH0: 'Hello World'
15:03:53.420 INFO Current text value on blockchain ETH1: 'Hello World'
15:03:53.454 INFO Current text value on blockchain ETH2: 'Hello World'
```

6.2.9 Scenario 9: Dishonest Intermediary

Description

This experiment is similar to Scenario 2. This time, we are manipulating the winning intermediary in different ways:

1. The intermediary does not execute the cross-blockchain call. The expected result is a fraud detection. For demonstration purposes, we configure the fraud detection period to 100 blocks.

2. The intermediary does not return the correct result. The expected behavior is a vote that the result is invalid.

3. The intermediary tries to cheat with the gas cost. The expected behavior is a vote that the result is valid but the intermediary has cheated.

Each experiment need to be done only once to show the correct detection of misconduct.

Result

The following listings show the different error constellations. On all listings, it can be seen that the intermediary has not received back its used tokens.

1. Lines 4 and 5 of Listing 6.11 show a different sequence than the previous listings. Since no result has been submitted, a fraud voting has to be initialized by the validator.

2. Line 6 of Listing 6.12 shows that the imported result does not match, since the validator voted 0 instead of 2.

3. Line 6 of Listing 6.13 shows that the imported result is valid, but the required gas is invalid, since the validator voted 1 instead of 2.

Listing 6.11: Log Extraction Illustrating the Non Execution of the Cross-Blockchain Transaction

```
1  21:33:51.894 INFO balance 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7: 99005600026733038921
2  ...
3  21:34:50.879 INFO New event 'NewBestOffer' on contract distributionContract on blockchain ETH0
       blockNumber: 7169 data: {"invocationId":{"_hex":"0x10"}}
4  21:44:46.711 INFO New event 'FraudVotingStarted' on contract distributionContract on blockchain
       ETH0 blockNumber: 7288 data: {"invocationId":{"_hex":"0x10"}}
5  21:44:46.711 INFO New event 'NewFraudVotingWinner' on contract distributionContract on blockchain
       ETH0 blockNumber: 7288 data: {"invocationId":{"_hex":"0x10"},
       "winner":"0x7295BB66b1f6e2b49203FDe339482AD8F86c5b50"}
6  21:46:31.896 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
       blockNumber: 7309 data: {"invocationId":{"_hex":"0x10"}}
7  21:46:32.338 INFO Result valid on blockchain ETH0: 'false', result status: 'false'
8  21:46:32.346 INFO balance 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7: 99005600026732966473
```

Listing 6.13: Log Extraction Illustrating the Import of Wrong Gas Values

```
21:53:46.892 INFO balance 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7: 99005600026732493694
...
21:56:24.189 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH1
     blockNumber: 6194 data: {"sender":"0x21574114b889Ea4Dac00d46b445324fdE3b56EA1",
     "resultId":{"_hex":"0x0f"}}
21:57:24.496 INFO Manipulating gas
21:57:29.236 INFO New event 'ResultAvailable' on contract distributionContract on blockchain ETH0
     blockNumber: 7440 data: {"sender":"0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7",
     "invocationId":{"_hex":"0x12"}}
21:58:14.919 INFO Voted 0x7295BB66b1f6e2b49203FDe339482AD8F86c5b50, invocationId: 18, value: 1
22:00:04.587 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
     blockNumber: 7471 data: {"invocationId":{"_hex":"0x12"}}
22:00:05.444 INFO Result value on blockchain ETH0: '0x00000000000000000000000000000000
     00000000000000000000000000000020 00000000000000000000000000000000
     0000000000000000000000000000000b 48656c6c6f20576f726c640000000000
     00000000000000000000000000000000', translated: Hello World
22:00:05.448 INFO balance 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7: 99005600026731959177
```

Listing 6.12: Log Extraction Illustrating the Import of a Wrong Result

```
21:47:19.338 INFO balance 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7: 99005600026732966473
...
21:49:59.175 INFO New event 'NewExecuteResult' on contract invocationContract on blockchain ETH1
     blockNumber: 6130 data: {"sender":"0x21574114b889Ea4Dac00d46b445324fdE3b56EA1",
     "resultId":{"_hex":"0x0e"}}
21:51:00.253 INFO Manipulating result
21:51:02.467 INFO New event 'ResultAvailable' on contract distributionContract on blockchain ETH0
     blockNumber: 7363 data: {"sender":"0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7",
     "invocationId":{"_hex":"0x11"}}
21:51:48.336 INFO Voted 0x7295BB66b1f6e2b49203FDe339482AD8F86c5b50, invocationId: 17, value: 0
21:53:37.848 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
     blockNumber: 7394 data: {"invocationId":{"_hex":"0x11"}}
21:53:37.980 INFO Result valid on blockchain ETH0: 'false', result status: 'true'
21:53:37.984 INFO balance 0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7: 99005600026732493694
```

6.2.10 Scenario 10: Dishonest Validator

Description

This experiment is similar to Scenario 5. However, one of the three validators behaves dishonestly and votes against the result. Since over 50% of the validators have voted for success, it is expected that the result is still marked as valid. This experiment has to be executed only once, as the result is independent of the exact validator who votes against the result. This means that the outcome of the experiment is always the same, regardless of whether validator one, two or three voted against the result, as long as the other two have voted correctly.

Result

Listing 6.14 shows the invalid voting of a dishonest validator (line 4) and the still correct voting result (line 9).

Listing 6.14: Log of an Invalid Vote of a Single Validator

```
22:00:18.425 INFO Current bool value on blockchain ETH1: 'false'
...
22:03:55.407 INFO New event 'ResultAvailable' on contract distributionContract on blockchain ETH0
     blockNumber: 7517 data: {"sender":"0xF23A7eb6EDF8D5Ca365C27a2E46D0E96857876b7",
     "invocationId":{"_hex":"0x13"}}
22:04:41.238 INFO Voted 0x7295BB66b1f6e2b49203FDe339482AD8F86c5b50, invocationId: 19, value: 0
22:04:41.446 INFO Voted 0xe87858a6278747C5B6D13d3D34b8944AC9227F31, invocationId: 19, value: 2
22:04:41.658 INFO Voted 0x535F9924928De7A6160cBDBCf9c9ade5Ad437809, invocationId: 19, value: 2
...
22:06:30.857 INFO New event 'CallFinished' on contract distributionContract on blockchain ETH0
     blockNumber: 7548 data: {"invocationId":{"_hex":"0x13"}}
22:06:31.353 INFO Current bool value on blockchain ETH1: 'true'
```

6.2.11 Scenario 11: Cost Calculation

Description

After the previous scenarios have tested the functionality, this scenario compares the cost of various functions by manually executing the transaction on the target chain for comparison with the prototype. For this, the gas cost per step is set to one, so that the steps can be read from the transaction logs or can be calculated by subtracting the account balance after the transaction from the account balance before the transaction. With the number of steps, a calculation with real numbers from the current Ethereum blockchain can be conducted by multiplying the current transaction prices with the given steps.

Since the required steps vary between different parameters, the cost of the setters and getters listed in Table 6.2 are determined. While we assume that the return values of the setters are not needed on the source chain, the return values of getters have to be imported into the source chain. For this, a setter method is called on the source chain for the manual step and a callback function is applied for the prototype.

Result

The results are summarized in Table 6.3. It is visible that registering and executing a call as well as posting the result will cause the highest gas cost and depends on the function signature as well as the parameter and the return value length. The worst case gas cost for the defined setters is

$$388568 + 72448 + 331818 + 309081 + 110267 + 93733 = 1305915 \text{ Steps}$$

and of the getters

$$354389 + 72448 + 306348 + 384982 + 110267 + 131353 = 1359787 \text{ Steps}$$

taking into account only one intermediary, one validator and no fees.

The total required gas is about 40 times higher than with manual invocations (Setters: $\frac{1305915}{33022} \approx 40$, Getters: $\frac{1359787}{33022} \approx 41$ times), assuming that the getter functions do not cost

Table 6.3: Required Steps per Transaction

Operation	Required Steps
`registerCall`, without callback, Setter	362,967-388,568
`registerCall`, with callback, Getter	354,325-354,389
`makeOffer`, first call	72,448
`makeOffer`, not a better offer	26,131
`makeOffer`, better offer	49,011
`executeCall`, Setter	290,667-331,818
`executeCall`, Getter	265,403-306,348
`postResult`, Setter (empty)	309,081
`postResult`, Getter	344,445-384,982
`vote`, first vote	110,267
`vote`, other votes	60,678
`finalize`, without callback	93,733
`finalize`, with callback	111,114-131,353
manual setter calls	26,983-33,022
manual getter calls	0

anything when invoked manually, as no transaction has to be executed on the blockchain and only the setter is required for the import. Looking more closely at `executeCall`, which is more or less the call of a setter function, it has alone an overhead of about 300,000 steps or by a factor of 9.

Considering the current gas prices, which are about 11 gwei per step[1], such a transaction costs about 0.012 Ether or \$3.24 (current market value of 1 Ether=\$270[2]) compared to about \$0.10 (=0.00037 Ether) for a manual invocation.

6.2.12 Scenario 12: Time Overhead Estimation

Description

As described in Section 5.3, there is a time overhead when executing the cross-blockchain call over this prototype. In this scenario, a comparison of the time spent between the prototype and manual execution is determined using Scenarios 1, 6, 7, and 8, as these scenarios differ from the number of cross-blockchain calls. The time required for the prototype is calculated as the difference of the block numbers between the functions `registerCall` and `finalize`, while the number of blocks for the manual execution can be estimated by ignoring different block creation times and using the value of `waitingBlocks` for each required invocation (e.g., `executeCall` and callback).

Furthermore, the amount of time spent is estimated by using different values for `waitingBlocks` and `blocksPerPhase`. While the number of `waitingBlocks` is

[1] `https://etherscan.io/gastracker`, accessed 2019-05-28
[2] `https://coinmarketcap.com/currencies/ethereum/`, accessed 2019-05-28

important for consistency and should not be set to low, the amount of `blocksPerPhase` guarantees enough time to get fair offers and correct voting results. We use different combinations for `waitingBlocks` and `blocksPerPhase` for the evaluation, where we set the number of `waitingBlocks` to 30, 50 and 100 and the number of `blocksPerPhase` to 5, 10 and 30.

By using a local test network, the time between sending a transaction and including it into a block, i.e., the wait time in the transaction pool, is missing, since a local test chain usually includes the transaction in the next block. Therefore, Scenario 1 of this experiment is also executed on an official Ethereum test network called Rinkeby, where the contracts are deployed two times on the same blockchain to simulate instead of using explicitly two different blockchains. This makes no difference in the evaluation process, since the logic remains the same and the contracts are not aware of the location of the other blockchain. The transaction on the official test network is executed 3 times and the average number of blocks are used. Similar to the local test network, Rinkeby uses the PoA consensus algorithm and has a block mining time of about 15 seconds. The retrieved numbers of block cannot be compared to the local numbers due to different block mining times and their deviations. However, the impact of the transaction pool can be measured by comparing with optimal block numbers.

An execution of all scenarios on the test network is not feasible in the given timeframe, since the block mining time is higher than the local test networks. Therefore, we decided to only evaluate the base case on an official network and run the other scenarios only on the local test network and estimate the execution based on following calculation:

Taking into account the 11 gwei per step used in Scenario 11, then the transaction is included in the block on average after 18 seconds[3] in the official Ethereum blockchain (1-2 blocks). This is only one additional block per transaction and therefore negligible for the experiment. Of course, if the transactions are executed with less gas or the blockchain is not synced completely, the duration increases as the transactions stay longer in the transaction pool, but at least if less gas is provided the calculated cost from Scenario 11 decreases.

Result

Table 6.4 shows the results for Scenario 1, Table 6.7 for Scenario 6, Table 6.8 for Scenario 7, and Table 6.9 for Scenario 8.

The following list explains the estimation for the manual executions (denoted as m in the tables):

- Scenario 1: The `setText` function is called. The call is considered valid after `waitingBlocks` blocks. The return value is not required, so no further transactions have to be invoked.

[3]`https://etherscan.io/gasTracker`, accessed 2019-06-09

- Scenario 6: The `getText` function is called. The returned value is considered valid after `waitingBlocks` blocks and has to be imported into the source chain. Since the continuation method is invoked at the end of the cross-blockchain call and therefore the validity of the continuation method is not checked, no blocks have to be considered.

- Scenario 7: A `getText` function is called via *ETH2*. Since *ETH2* forwards the call without any additional functionality, *ETH1* can be invoked directly. Since the call now corresponds to Scenario 6, the same estimate applies.

- Scenario 8: First, *ETH2* is invoked, which triggers the `setText` function on *ETH1*. To retrieve the parameters for `setText`, they have to be queried from *ETH2*. This requires `waitingBlocks` blocks to consider these values as valid. Then, `setText` can be called, which requires `waitingBlocks` blocks to consider the call as valid (as in Scenario 1).

  Subsequently, `getText` is invoked on *ETH1* by *ETH2*. As in Scenario 6, this call requires `waitingBlocks` blocks. This time, however, the value is imported into *ETH2*, which is not the start chain. Therefore, `waitingBlocks` blocks have to be waited to consider this import as valid. Only then is the value imported into *ETH0*, which can be ignored with the argument given in Scenario 6.

  Therefore, the total is $waitingBlocks \cdot 4$ blocks, once to retrieve the parameters, once for `setText` and `getText` on *ETH1* and once for the continuation method on *ETH2*.

We would like to point out that increasing the block configuration does not increase the required number of blocks by the same factor, since other blockchains have different block mining times and thus the time to reach the next block differs between the blockchains. For example, Table 6.4 shows an increase of 11 blocks with a configuration of 30 `waitingBlocks` when the parameter `blocksPerPhase` is increased from 5 to 10. This has not been taken into account when estimating the manual block time.

Example 6.1 *At the given block mining times of 5 (ETH0), 6 (ETH1), and 4 (ETH2) seconds, the blockchains generate 12 (ETH0), 10 (ETH1) and 15 (ETH2) blocks in one minute. For example, to reach 30 blocks, 2.5 (ETH0), 3 (ETH1) and 2 (ETH2) minutes are needed. When the number of blocks is measured on ETH0, then 36 (ETH1) or 24 (ETH2) blocks are generated in the time when 30 blocks are reached on the other chain.*

The result shows an increase of factor 5-7 for a single cross-blockchain call, depending on the number of `blocksPerPhase`. Given the current blockchain time of about 15 seconds per block, the transaction takes 45 minutes if `waitingBlocks` is set to 30 and `blocksPerPhase` is set to 10, compared to 7.5 minutes if invoked manually. If, as in Scenario 7, two cross-blockchain calls can be merged into one manual call, the factor increases to 10-12. However, in these cases, the blockchain call may be adjusted as well to remove the unnecessary step. For example, in Scenario 7 the call via *ETH2* is not required and can be removed there as well.

Example 6.2 ***Cost-Time Tradeoff***
As discussed in the description, depending on the gasprice the transactions may stay longer in the transaction pool. Assume that the cross-blockchain call of Scenario 1 is executed only with 2 gwei per step. Then it takes 98 seconds or about 7 blocks[a] *to include a transaction into the block, but the execution cost decreases to $0.58. Thus, the amount of blocks increases by about 35 (the call to* `makeOffer` *has no influence since it does not control any phase) and the manual execution by about 7 blocks. The factor gets slightly better due to the larger relative increase of the denominator. Figure 6.4 visualizes the estimated tradeoff.*

[a] `https://etherscan.io/gasTracker`, accessed 2019-06-09

Figure 6.2 illustrates the distribution of blocks spent in each phase of the cross-blockchain call for Scenario 1 using 30 `waitingBlocks` and 5 `blocksPerPhase` and Figure 6.3 for Scenario 8. The figures show which phases use which parameter (OfferPhase and VotingPhase use `blocksPerPhase`, the others use `waitingBlocks`). Furthermore, the different block mining times of the blockchains are visible in Scenario 1, where the TransactionPhase takes 4% longer than the other phases. The TransactionPhase in Scenario 8 takes more than 50% of the time because it has to wait until the final result of the second cross-blockchain call is available.

Considering the values given in Table 6.6 for Scenario 1 on an official test network, the transaction pool has a nearly consistent and negligible impact on the query of about 1 block. As comparison, Table 6.5 lists the optimal times for Rinkeby based on the formula

$$5 \cdot waitingBlocks + 2 \cdot blocksPerphase + 4 \tag{6.1}$$

where the summand $+4$ is derived from the one block offsets until the next action is triggered (import of call, import of result, first vote, finalize) and the other values are based on the discussion in Chapter 5. For most of the calls, the number of blocks stays the same, only for some calls, there has been an impact on the block count. This can depend not only on the size of the transaction pool, but also on the arrival of blocks due to network latency. For example, by $blocksPerPhase = 10$ and $waitingBlocks = 30$, there was an outlier with 183 instead of 177 and for $blocksPerPhase = 10$ and $waitingBlocks = 30$, there was an outlier with 630 instead of 567.

Table 6.4: Time Consumption for Scenario 1 on Local Network (in Blocks)

blocksPerPhase	waitingBlocks 30	50	100
5	168	273	533
10	179	282	542
30	218	323	582
m	30	50	100

Table 6.5: Optimal Time Consumption for Scenario 1 on Rinkeby (in Blocks)

blocksPerPhase	waitingBlocks 30	50	100
5	164	264	514
10	174	274	524
30	214	314	564
m	30	50	100

Table 6.6: Average Time Consumption for Scenario 1 on Rinkeby (in Blocks)

blocksPerPhase	waitingBlocks 30	50	100
5	167	267	517
10	179	277	527
30	217	317	588
m	30	50	100

Table 6.7: Time Consumption for Scenario 6 (in Blocks)

blocksPerPhase	waitingBlocks 30	50	100
5	168	272	532
10	178	282	542
30	218	322	583
m	30	50	100

Table 6.8: Time Consumption for Scenario 7 (in Blocks)

blocksPerPhase	waitingBlocks 30	50	100
5	299	482	942
10	317	500	960
30	388	572	1,033
m	30	50	100

Table 6.9: Time Consumption for Scenario 8 (in Blocks)

blocksPerPhase	waitingBlocks 30	50	100
5	441	712	1,392
10	466	738	1,417
30	570	843	1,522
m	120	200	400

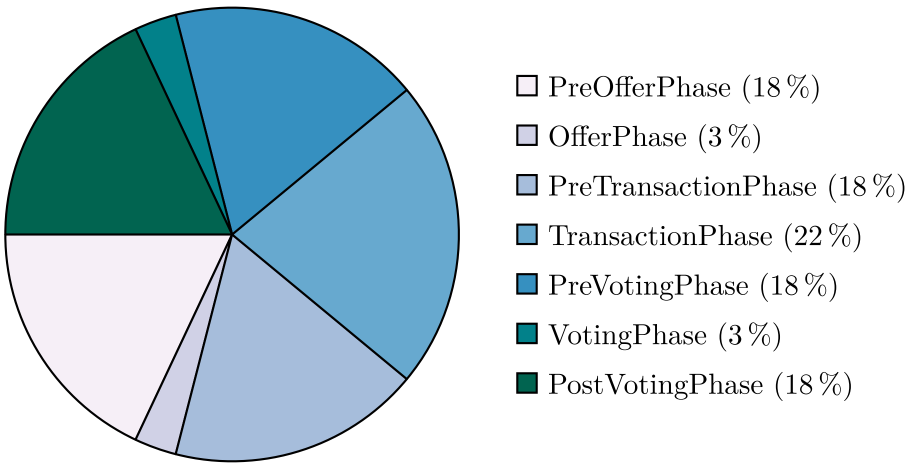


Figure 6.2: Time Distribution per Phase for Scenario 1 (waitingBlocks=30, blocksPerPhase=5)

6.2.13 Summary of Results

The first scenarios focus on the functional correctness of the prototype. For this, the parallel execution of cross-blockchain calls, the recursive invocation as well as the continuation of a cross-blockchain call and the handling of dishonest parties were evaluated.

The last two scenarios discusses the cost and performance of the prototype. The results have shown that the costs increase by a factor of 40 and the performance by a factor of 5-7, depending on the configuration, the transaction pool and the network latency of the blockchain node.

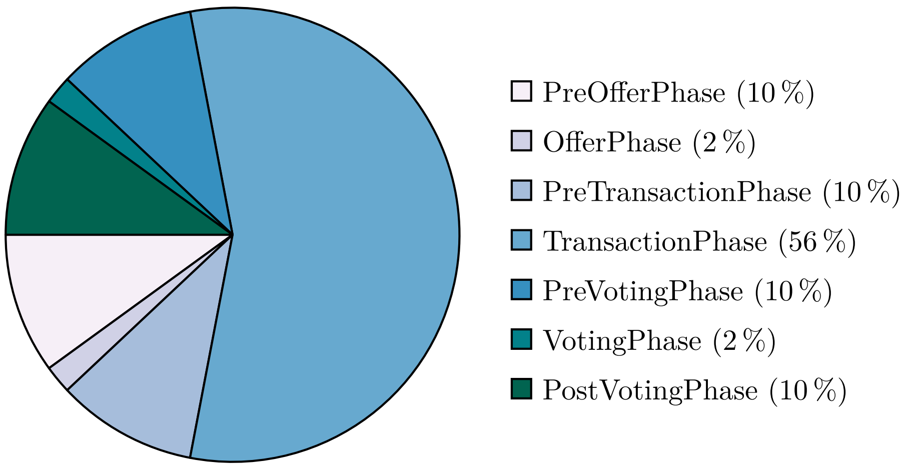


Figure 6.3: Time Distribution per Phase for Scenario 7 (waitingBlocks=30, blocksPerPhase=5)

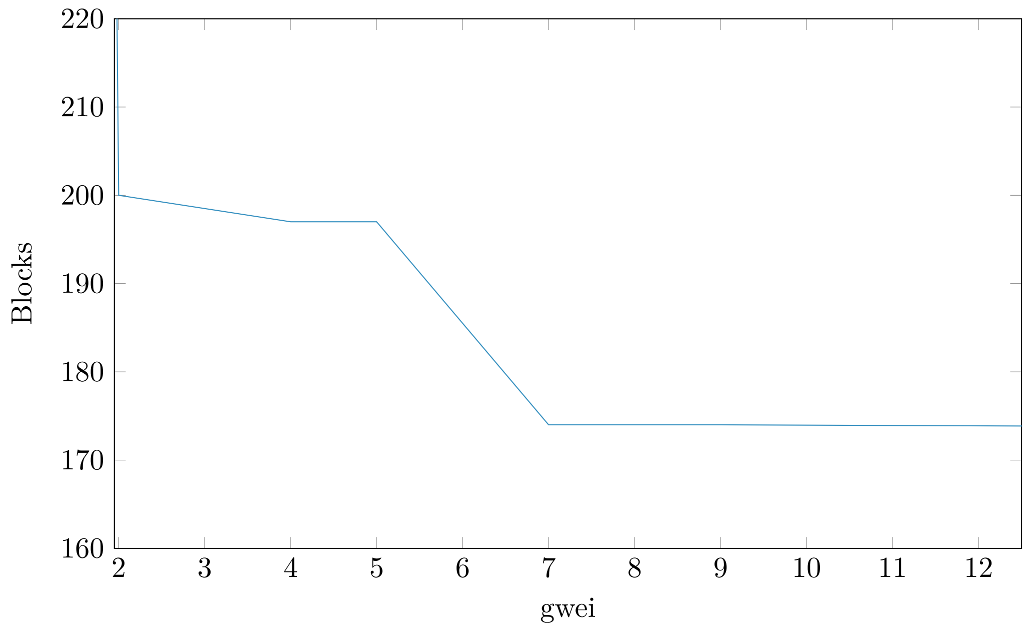


Figure 6.4: Estimated Time-Cost Tradeoff for Scenario 1 (waitingBlocks=30, blocksPerPhase=5)

CHAPTER 7

Conclusion & Future Work

This thesis aims to improve the interoperability of blockchains. Based on the concepts of state-of-the-art methods of cross-blockchain interaction, a method for invoking smart contracts across blockchains has been developed. It takes into account the correct execution of a transaction, the concurrency and consistency of blockchains and the payment of transaction fees and rewards. The method supports not only a single cross-blockchain call, but also the recursive invocation of cross-blockchain calls and the continuation of a function after receiving a result.

A prototype has been implemented, supporting all blockchains based on the EVM that use the Solidity compiler. The functionality of the prototype was evaluated with several tests where results had a performance overhead by a factor of 6 and a cost overhead by a factor of 40.

7.1 Discussion of Research Questions

This section provide answers to the three research questions introduced in Chapter 1.

1. **Which blockchain technologies are suitable candidates for implementing cross-blockchain smart contract calls? Which methods for interacting between blockchains exist? Can they be used as a basis for invoking smart contracts?**

   This question mainly focuses on providing an overview of existing technologies. In Chapter 2, we discussed the fundamentals of blockchains and the characteristics of common cryptocurrencies. We discovered that only Eos, Ethereum and Tron are suitable candidates for implementing a prototype. As stated in Section 4.1, we chose Ethereum because of thorough discussions in existing literature.

Chapter 3 presented state-of-the-art methods that deal with blockchain interoperability. We discussed atomic cross-chain swaps, off-chain payment networks (Lightning and Raiden) and cross-blockchain tokens (Metronome and DeXTT). While many of the concepts discussed use signatures from both parties that cannot be used with smart-contracts because a private key cannot be stored in publicly visible storage, the concept of intermediaries and validators introduced by Metronome and DeXTT was used as basis for our work.

2. **How can smart contracts be called across blockchains?**
   In Chapter 4, we introduced our design for invoking smart contracts across blockchains and described the concepts of our prototype implementation in Chapter 5.

   We split the design into two parts, the invocation of a smart contract and the receiving of the result data. While in the invocation part the important concept was to ensure the payment for the transaction and to select only one intermediary that executes the transaction to reduce the gas cost, the main issue in receiving the result was the verification of the correct execution.

3. **How does the prototype perform in terms of cost and performance?**
   The evaluation in Chapter 6 shows an increase in terms of cost and performance. While the cost increases by a factor of 40, the time in the given configurations only increased by an average factor of 6. If set in absolute terms, a transaction cost \$3.24 and takes about 45 minutes (waitingBlocks=30 and blocksPerPhase=10) compared to \$0.10 and 7.5 minutes if invoked manually.

7.2 Future Work

This section reviews some of the issues that remain open for future research and highlights possibilities to extend the prototype.

7.2.1 Game Theory: Voting and Offering Process

As has already been discussed in Chapter 4, the incentives to vote honestly are only given due to the dPoS protocol and should be improved. For this purpose, multiple scenarios can be developed and evaluated. For example, the consequences of hiding votes before the end of the voting phase or the use of Merkle trees for valid votings, as used in Metronome, may be discussed.

In addition, the incentives for receiving a fair and good offer may be evaluated as they may vary according to the duration of the offer phase and the chosen algorithm to end the offer phase, as discussed in Section 4.3.5.

7.2.2 Adding Support for Other Blockchains

The implemented prototype is built for languages which use the EVM and support the Solidity compiler. The prototype can be extended to support other blockchains such as Eos by converting the smart contracts for these blockchains and by adding functional support to the intermediaries and validators for the new blockchains. This involves analyzing the properties of the added blockchain, like handling blockchains that do not use gas as a concept for transaction cost.

7.2.3 Extending the Functionality

As stated in Section 5.6, the prototype has some limitations. One of them is the missing implementation of token transfers that would allow callers to invoke functions that require tokens.

Furthermore, the announcement and propagation of the return value of cross-blockchain calls can be improved. Among other things, developers can be given the opportunity to decide which cross-blockchain call they want to wait for instead of using the last call. In addition, the necessary keyword can be omitted as return value.

Moreover, the party calling the finalization step can be chosen in an optimized way. For example, it either should be limited to winning parties, or an additional reward could be introduced for calling the finalize function.

7.2.4 Reduce the Cost of the Implementation

The evaluation of the costs resulted in a cost increase by a factor of 40. Since this might be unsatisfactory and the costs can be considered too expensive for certain practical scenarios, we are looking for a way to reduce the cost of cross-blockchain calls. Possible options include the use of off-chain transactions or the optimization of the source code of the implementation.

List of Figures

List of Tables

Listings

Acronyms

ABI Application Binary Interface. 14

DAO Decentralized Autonomous Organization. 16

DeXTT Deterministic Cross-Blockchain Token Transfers. 24, 26, 28, 30, 33, 34, 37, 39, 96

dPoS delegated Proof of Stake. 10, 17, 18, 44, 48, 96

EVM Ethereum virtual machine. 14, 17, 95, 97

HTLC hashed time-lock contract. 19, 23

PoA Proof of Authority. 10, 48, 89

PoI Proof of Intent. 26–28

PoS Proof of Stake. 10, 17, 18, 25

PoW Proof of Work. 10, 16, 17, 35

UTXO unspent transaction output. 8, 11–13, 21, 22

Bibliography

[ABC17] Nicola Atzei, Massimo Bartoletti, and Tiziana Cimoli. A survey of attacks on ethereum smart contracts (sok). In Matteo Maffei and Mark Ryan, editors, *Principles of Security and Trust*, pages 164–186, Berlin, Heidelberg, 2017. Springer Berlin Heidelberg.

[Ash18] Nathan Ashworth. Five examples of real-world uses for smart contracts. `https://cryptoinsider.com/examples-real-world-uses-smart-contracts/`, 2018. [Online; accessed 2019-04-13].

[ato17] Decred-compatible cross-chain atomic swapping. `https://github.com/decred/atomicswap`, 2017. [Online; accessed 2019-03-24].

[Aut18] Autonomous Software. Metronome. `https://www.metronome.io/download/owners_manual.pdf`, 2018. [Online; version 0.988; accessed 2019-02-24].

[Aut19] Autonomous Software. Metronome cross-chain token transfer. `https://github.com/autonomoussoftware/documentation/blob/master/validatordocument/validatordocument.md`, 2019. [Online; version 0.905; accessed 2019-03-24].

[Ban18] Eric Banisadr. How $800k evaporated from the powh coin ponzi scheme overnight. `https://blog.goodaudience.com/how-800k-evaporated-from-the-powh-coin-ponzi-scheme-overnight-1b025c33b530`, 2018. [Online; accessed 2019-04-13].

[bar17] barterdex - atomic swap decentralized exchange of native coins. `https://github.com/SuperNETorg/komodo/wiki/barterDEX-Whitepaper-v2`, 2017. White Paper. [Online; accessed 2019-02-24].

[BDJS17] Lorenz Breidenbach, Phil Daian, Ari Juels, and Emin Gün Sirer. An in-depth look at the parity multisig bug. `http://hackingdistributed.com/2017/07/22/deep-dive-parity-bug/`, 2017. [Online; accessed 2019-04-13].

[BFS+18] Michael Borkowski, Philipp Frauenthaler, Marten Sigwart, Taneli Hukkinen, Oskar Hladký, and Stefan Schulte. Deterministic witnesses for claim-first transactions. http://dsg.tuwien.ac.at/staff/mborkowski/pub/tast/tast-white-paper-3.pdf, 2018. White Paper, Technische Universitat Wien. [Online; accessed 2019-02-24].

[bita] Atomic swap. https://en.bitcoin.it/wiki/Atomic_swap. [Online; accessed 2019-04-13].

[bitb] Bitcoin cash. https://www.bitcoincash.org/index.html. [Online; accessed 2019-03-27].

[blo18] block.one. Eos.io technical white paper v2. https://github.com/EOSIO/Documentation/blob/master/TechnicalWhitePaper.md, 2018. White Paper. [Online; accessed 2019-03-23].

[BMM+20] Stefano Bistarelli, Gianmarco Mazzante, Matteo Micheletti, Leonardo Mostarda, and Francesco Tiezzi. Analysis of ethereum smart contracts and opcodes. In *Advanced Information Networking and Applications*, pages 546–558. Springer International Publishing, 2020.

[BRMS18a] Michael Borkowski, Christoph Ritzer, Daniel McDonald, and Stefan Schulte. Caught in chains: Claim-first transactions for cross-blockchain asset transfers. http://dsg.tuwien.ac.at/staff/mborkowski/pub/tast/tast-white-paper-2.pdf, 2018. White Paper, Technische Universitat Wien. [Online; accessed 2019-02-24].

[BRMS18b] Michael Borkowski, Christoph Ritzer, Daniel McDonald, and Stefan Schulte. Towards atomic cross-chain token transfers: State of the art and open questions within tast. http://dsg.tuwien.ac.at/staff/mborkowski/pub/tast/tast-white-paper-1.pdf, 2018. White Paper, Technische Universitat Wien. [Online; accessed 2019-02-24].

[BSCG+14] Eli Ben-Sasson, Alessandro Chiesa, Christina Garman, Matthew Green, Ian Miers, Eran Tromer, and Madars Virza. Zerocash: Decentralized anonymous payments from bitcoin. In *2014 IEEE Symposium on Security and Privacy*, pages 459–474, 2014.

[BXW19] Dilum Bandara, Xiwei Xu, and Ingo Weber. Patterns for blockchain migration, 06 2019.

[Car] Cardano Foundation. Cardano settlement layer documentation. https://cardanodocs.com/introduction/. [Online; accessed 2019-03-23].

[CD17] Ignacio Cascudo and Bernardo David. Scrape: Scalable randomness attested by public entities. In *Applied Cryptography and Network Security*, pages 537–556. Springer International Publishing, 2017.

[Cle18] Clearmatics. Ion stage 2: Cross-chain smart contract development. https://medium.com/clearmatics/ion-stage-2-cross-chain-smart-contract-development-part-4-d9bcc5abe2ff, 2018. [Online; accessed 2019-06-10].

[CM18] Brad Chase and Ethan MacBrough. Analysis of the XRP ledger consensus protocol. *CoRR*, abs/1802.07242, 2018.

[col] Colored coins. https://en.bitcoin.it/wiki/Colored_Coins. [Online; accessed 2019-03-27].

[Com18] Christina Comben. What are blockchain confirmations and why do they matter? https://coincentral.com/blockchain-confirmations/, 2018. [Online; accessed 2019-05-07].

[DD18] Evan Duffield and Daniel Diaz. Dash: A payments-focused cryptocurrency. https://github.com/dashpay/dash/wiki/Whitepaper, 2018. White Paper. [Online; accessed 2019-03-22].

[dex19] Dextt: Deterministic cross-blockchain token transfers. https://github.com/pantos-io/dextt-prototype, 2019. [Online; accessed 2019-03-25].

[DGHK17] Thomas Dickerson, Paul Gazzillo, Maurice Herlihy, and Eric Koskinen. Adding concurrency to smart contracts. In *ACM Symposium on Principles of Distributed Computing*, (PODC '17), pages 303–312. ACM, 2017.

[Etha] Ethereum Foundation. Common patterns. https://solidity.readthedocs.io/en/v0.5.6/common-patterns.html. [Online; accessed 2019-05-06].

[Ethb] Ethereum Foundation. Contracts. https://solidity.readthedocs.io/en/v0.5.6/contracts.html. [Online; accessed 2019-03-21].

[Ethc] Ethereum Foundation. A next-generation smart contract and decentralized application platform. https://github.com/ethereum/wiki/wiki/White-Paper. White Paper. [Online; accessed 2019-02-24].

[Her18] Maurice Herlihy. Atomic cross-chain swaps. In *ACM Symposium on Principles of Distributed Computing*, (PODC '18), pages 245–254. ACM, 2018.

[Her19] Maurice Herlihy. Blockchains from a distributed computing perspective. *Commun. ACM*, 62(2):78–85, 2019.

[kin16] Post-mortem investigation. https://www.kingoftheether.com/postmortem.html, 2016. [Online; accessed 2019-04-13].

[kom18] Komodo: advanced blockchain technology, focused on freedom. https://komodoplatform.com/wp-content/uploads/2018/06/Komodo-Whitepaper-June-3.pdf, 2018. White Paper. [Online; accessed 2019-03-24].

[KRDO17] Aggelos Kiayias, Alexander Russell, Bernardo David, and Roman Oliynykov. Ouroboros: A provably secure proof-of-stake blockchain protocol, 2017.

[Lit] Litecoin Foundation. Litecoin. https://litecoin.org/. [Online; accessed 2019-03-22].

[LKKY05] Sung-Woon Lee, Hyun-Sung Kim, and Yoo Kee-Young. Efficient nonce-based remote user authentication scheme using smart cards. *Applied Mathematics and Computation*, 167:355–361, 2005.

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

[Maz16] David Mazières. The stellar consensus protocol: A federated model for internet-level consensus. https://www.stellar.org/papers/stellar-consensus-protocol.pdf, 2016. [Online; accessed 2019-03-23].

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system. http://bitcoin.org/bitcoin.pdf, 2008. [Online; accessed 2019-02-24].

[NBF+16] Arvind Narayanan, Joseph Bonneau, Edward Felten, Andrew Miller, and Steven Goldfeder. *Bitcoin and Cryptocurrency Technologies: A Comprehensive Introduction.* Princeton University Press, Princeton, NJ, USA, 2016.

[omn] Omni layer. https://www.omnilayer.org. [Online; accessed 2019-03-27].

[PD16] Joseph Poon and Thaddeus Dryja. The bitcoin lightning network: Scalable off-chain instant payments. https://lightning.network/lightning-network-paper.pdf, 2016. [Online; accessed 2019-03-26].

[raia] Raiden network specification. https://raiden-network-specification.readthedocs.io/en/latest/. [Online; accessed 2019-03-26].

[raib] What is the raiden network? https://raiden.network/101.html. [Online; accessed 2019-03-26].

[Reu18] Arseny Reutov. Predicting random numbers in ethereum smart contracts. https://blog.positive.com/predicting-random-numbers-in-ethereum-smart-contracts-e5358c6b8620, 2018. [Online; accessed 2019-05-06].

[RSA78] R. L. Rivest, A. Shamir, and L. Adleman. A method for obtaining digital signatures and public-key cryptosystems. *Commun. ACM*, 21(2):120–126, 1978.

[Sha] Gordon Shawn. What is ripple? `https://bitcoinmagazine.com/guides/what-ripple/`. [Online; accessed 2019-03-23].

[SN17] Matthew Spoke and Nuco Engineering Team. Aion: Enabling the decentralized internet. `https://aion.network/media/en-aion-network-technical-introduction.pdf`, 2017. White Paper. [Online; accessed 2019-06-10].

[Sun18] Flora Sun. A survey of consensus algorithms in crypto. `https://medium.com/@sunflora98/a-survey-of-consensus-algorithms-in-crypto-e2e954dc9218`, 2018. [Online; accessed 2019-03-20].

[SYB] David Schwartz, Noah Youngs, and Arthur Britto. The ripple protocol consensus algorithm. `https://ripple.com/files/ripple_consensus_whitepaper.pdf`. White Paper. [Online; accessed 2019-03-23].

[Ten19] Tendermint Inc. Cosmos – a network of distributed ledgers. `https://cosmos.network/resources/whitepaper`, 2019. White Paper. [Online; accessed 2019-06-10].

[Tik18] Sergei Tikhomirov. Ethereum: State of knowledge and research perspectives. In Abdessamad Imine, José M. Fernandez, Jean-Yves Marion, Luigi Logrippo, and Joaquin Garcia-Alfaro, editors, *Foundations and Practice of Security*, pages 206–221, Cham, 2018. Springer International Publishing.

[TRO18] TRON Foundation. Tron: Advanced decentralized blockchain platform 2.0. `https://tron.network/static/doc/white_paper_v_2_0.pdf`, 2018. White Paper. [Online; accessed 2019-03-23].

[TS16] Florian Tschorsch and Björn Scheuermann. Bitcoin and beyond: A technical survey on decentralized digital currencies. *IEEE Communications Surveys Tutorials*, 18(3):2084–2123, 2016.

[VB15] Fabian Vogelsteller and Vitalik Buterin. Eip 20: Erc-20 token standard. `https://eips.ethereum.org/EIPS/eip-20`, 2015. [Online; accessed 2019-03-20].

[Woo17] Gavin Wood. Polkadot: Vision for a heterogeneous multi-chain framework, draft 1. `https://polkadot.network/PolkaDotPaper.pdf`, 2017. White Paper. [Online; accessed 2019-06-10].

[ZBZ11] Yunhui Zheng, Tao Bao, and Xiangyu Zhang. Statically locating web application bugs caused by asynchronous calls. In *Proceedings of the 20th International Conference on World Wide Web*, WWW '11, pages 805–814, New York, NY, USA, 2011. ACM.

[ZSJ$^+$19] Alexei Zamyatin, Nicholas Stifter, Aljosha Judmayer, Philipp Schindler, Edgar R. Weippl, and William J. Knottenbelt. *A Wild Velvet Fork Appears! Inclusive Blockchain Protocol Changes in Practice*, volume 10958 of *Lecture Notes in Computer Science*. Springer, 2019.